DOCUMENT RESUME

ED 390 378 IR 017 597

AUTHOR	Madison, Sandra Kay
TITLE	A Study of College Students' Construct of Parameter Passing Implications for Instruction.
PUB DATE	Dec 95
NOTE	323p.; Ed.D. Dissertation, University of Wisconsin-Milwaukee.
PUB TYPE	Dissertations/Theses - Doctoral Dissertations (041) -- Tests/Evaluation Instruments (160)
EDRS PRICE	MF01/PC13 Plus Postage.
DESCRIPTORS	*College Students; Computer Literacy; Computer Science Education; Computer System Design; *Concept Formation; Higher Education; Interviews; *Introductory Courses; *Knowledge Representation; *Programmers; *Programming; Programming Languages
IDENTIFIERS	*Parameter Passing

ABSTRACT

Parameter passing is the mechanism by which various program modules share information in a complex program; this paper was a study of novice programmers' understanding of the parameter construct. The bulk of the data was collected from interviews with eight college students enrolled in a state university introductory computer programming course. Observations of the programming instruction, interviews with the course's instructor, and related student work provided additional data. Results revealed that the natural-language meaning of some computer terminology caused problems and most students' understanding of the parameter construct was fragile at the conclusion of their introductory course. Furthermore, it found that students' procedural knowledge (ability to construct modular programs that incorporate parameters) generally surpassed their conceptual knowledge (understanding of the parameter construct and of programs that incorporate the construct). The study also indicated that students who harbored fundamental misconceptions of the parameter process could, by making seemingly innocuous adjustments to procedure heading lines, construct programs that produced the correct answer. Concrete representations, collaboration in a structured laboratory environment, focused completion-type exercises, and elaboration appeared to foster success. Twenty-two appendices provide consent forms, a background survey form, excerpts from the literature, laboratory worksheets, assignments, interview protocols, and programming tasks. (Contains 53 references.) (Author/AEF)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION
CENTER (ERIC)
☐ This document has been reproduced as received from the person or organization originating it.
☐ Minor changes have been made to improve reproduction quality.
● Points of view or opinions stated in this document do not necessarily represent official OERI position or policy.

ED 390 378

A STUDY OF COLLEGE STUDENTS' CONSTRUCT OF PARAMETER PASSING

IMPLICATIONS FOR INSTRUCTION

by

Sandra Kay Madison

A Dissertation Submitted in
Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy

Urban Education

at

The University of Wisconsin-Milwaukee

December 1995

PERMISSION TO REPRODUCE THIS
MATERIAL HAS BEEN GRANTED BY
Sandra Madison

TO THE EDUCATIONAL RESOURCES
INFORMATION CENTER (ERIC)

IR017597

BEST COPY AVAILABLE

A STUDY OF COLLEGE STUDENTS' CONSTRUCT OF PARAMETER PASSING

IMPLICATIONS FOR INSTRUCTION

by

Sandra Kay Madison

A Dissertation Submitted in
Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy

Urban Education

at

The University of Wisconsin-Milwaukee

December 1995

Henry S Kepner Jr 16 August 1995
Major Professor (Signature) Date

Graduate School Approval Date

A STUDY OF COLLEGE STUDENTS' CONSTRUCT OF PARAMETER PASSING IMPLICATIONS FOR INSTRUCTION

by

Sandra Kay Madison

The University of Wisconsin-Milwaukee, 1995
Under the Supervision of Henry S. Kepner, Jr.

After reaching a peak in the early 1980s, student interest in computer science and information systems has declined dramatically. When coupled with increased industry demand for information systems professionals and higher than average salaries, the decline is especially troublesome. The Computer Information Systems program at a midwestern state university mirrors the problems reported by other sources. Attrition in the programming courses is high, and students, especially those from other disciplines taking the initial programming class as a service course, are often less than successful. Thus it is important to discover the factors of the computer programming course that dishearten students and discourage them from pursuing a computing career.

Parameter (argument) passing is the mechanism by which various program modules share information in a complex program. Without parameters, there is no genuine programming. Nevertheless, the computer programming instruction literature reveals a virtual absence of attention to the construct. This was a study of novice programmers' understanding of the parameter construct.

The study used a multi-case qualitative design. The bulk of the data was collected from interviews with eight college students enrolled in a state university introductory computer programming course. Observations of the programming instruction, interviews with the course's instructor, and related student work provided additional data.

Among other findings, the study revealed that the natural-language meaning of some computer terminology caused problems and that most students' understanding of the parameter construct was fragile at the conclusion of their introductory course. Furthermore, it found that students' procedural knowledge (ability to construct modular programs that incorporate parameters) generally surpassed their conceptual knowledge (understanding of the parameter construct and of programs that incorporate the construct). The study also divulged that students who harbored fundamental misconceptions of the parameter process could, by make seemingly innocuous adjustments to procedure heading lines, construct programs that produced the correct answer. Moreover, concrete representations, collaboration in a structured laboratory environment, focused completion-type exercises, and elaboration appeared to foster success.

Henry S Kepner Jr 16 August 1995

Major Professor (Signature) Date

© Copyright by Sandra K. Madison, 1995

All Rights Reserved

ACKNOWLEDGEMENTS

I have often remarked that if this degree were divided among all the people who helped make it happen, the pieces would be minuscule. I thank each one of them, and I hope I do not offend those who are not named individually.

I first thank my husband, Bob, and my daughters, Nicole and Amy, for their support. Eight years ago when I undertook this quest, they were forced to redefine the words "wife" and "mother." I am indebted to my good friends, David and Lee Allen and William and Susanne Gay, who unhesitatingly shared their homes and their lives during the many semesters of my commute.

I am grateful for the support of my professional colleagues. I especially thank William Thompson who gave me my start in education, Emery Babcock who encouraged me to develop my "educatorship," my colleague (known as Professor Collier) who pilot tested some of the instruments for me, and my department chairperson, William Wresch, whom I regard as my mentor. I am particularly indebted to the colleague, known throughout the paper as Dr. Greene, who allowed me to research in his class.

I thank my committee members, Derek Nazareth, Donald Neuman, and John Zahorik for their support and encouragement. Without that support and encouragement, I surely would have faltered. I thank my advisor, Henry Kepner, who supervised this project and guided my doctoral program from its inception.

Finally, I thank the wonderful students who served as the subjects in the study. They willingly shared their time and their thoughts, recognizing that their only reward would be the knowledge they helped improve instruction for future programming students.

Table of Contents

Approval Page ii

Abstract iii

Acknowledgements vi

Table of Contents viii

Lists of Figure and Tables x

Chapter I. INTRODUCTION 1
- Problem 1
- Factors of Success 4
- Cognitive Style 6
- Technical Information 8
- Purpose of the Study 19

Chapter II. LITERATURE REVIEW 20
- Framework: Constructivism 20
- Theoretical Bases of the Study 22
- Related Research Literature 37

Chapter III. METHODOLOGY 53
- Introduction 53
- General Design 54
- Setting 57
- Subjects and Sampling 58
- Data Collection Procedures 64
- Data Preparation 84
- Data Analysis 86
- Trustworthiness 89

Chapter IV. RESULTS 91
- Introduction to the Case Study Reports 91
- Case Study : Carol 92
- Case Study: Carl 100
- Case Study: Amy 107
- Case Study: Sammy 119
- Case Study: Sondra 135
- Case Study: Steve 149
- Case Study: Moe 168
- Case Study: Jason 180

Chapter V. DISCUSSION 197
Research Questions 197
Findings and Implications of the Study 225
Limitations of the Study and Future Directions 231

Bibliography 233

Appendices 246
Appendix A Instructor Informed Consent 246
Appendix B Participant Informed Consent 249
Appendix C Background Survey 252
Appendix D Excerpt from Inventory of Piaget's Developmental Tasks 255
Appendix E Laboratories 7 and 9 261
Appendix F Banner and Revised Banner Assignments 272
Appendix G Burger Queen Assignment 276
Appendix H Parameter-related Examination Questions 279
Appendix I Excerpt from Textbook Coverage of Parameters 284
Appendix J Participant Interview One Protocol 286
Appendix K Participant Interview Two Protocol 287
Appendix L Task 1 289
Appendix M Participant Interview Three Protocol 290
Appendix N Task 2 291
Appendix O Task 3 292
Appendix P Task 4 293
Appendix Q Instructor Interview One Protocol 294
Appendix R Instructor Interview Two Protocol 295
Appendix S Payroll Program Simulation 296
Appendix T Revised Leestma and Nyhoff Excerpt 298
Appendix U Payroll Program 302
Appendix V Task 4 Revised 304

VITA 311

List of Figures

Figure 1. Modular Pascal program that uses simple parameters 10

Figure 2. Depiction of Jason's variable parameter association 193

Figure 3. Depiction of parameters during execution of DisplayOutput 299

Figure 4. Depiction of the parameters as CalculateSalary begins execution 300

Figure 5. Depiction of the parameters as the execution of Calculate Salary completes 300

Figure 6. Depiction of parameters after execution of CalculateSalary 300

List of Tables

Table 1. Participant Profiles 63

Table 2 Descriptions of Variable 198

Table 3. Participants' Final Grades 222

CHAPTER I

INTRODUCTION

Problem

After reaching a peak in the early 1980s, student interest in computer science and information systems has declined dramatically (Lockheed & Mandinach, 1986). In 1983, more than 10% of American high school students chose computing fields as their intended college major; however, this interest dropped to 7% by 1985 (Lockheed & Mandinach, 1986) and continued to plummet during the early 1990s (Mawhinney, Cale, & Callaghan, 1990).

Moreover, national studies show that female and minority representation in computer-related majors echoes the pattern of women and minorities in mathematics and science; generally both groups are under-represented. Computing, like its sister disciplines, has traditionally been the domain of Euro-American males. The American Association of University Women (1992) report, *How Schools Shortchange Girls*, states the situation succinctly:

> A well-educated work force is essential to the country's economic development, yet girls are systematically discouraged from courses of study essential to their future employability and economic well-being. Girls are being steered away from the very courses required for their productive participation in the future of America, and we as a nation are losing more than one-half of our human potential. By the turn of the century, two out of three new entrants into the work force will be women and minorities. This work force will have fewer and fewer decently paid openings for the unskilled. It will require strength in science, mathematics, and technology. (p. v)

Although Linn and Songer (1991) echo the concern with female under-representation in mathematics and the sciences, the authors contend that "by far the most serious issue . . . is that far too few students altogether are being attracted to mathematics, science, and technology" (p. 409). When the declining technology interest is coupled with increased industry demand for information systems professionals and higher than average salaries, it is especially troublesome (Mawhinney et al., 1990).

The Computer Information Systems (CIS) program at a midwestern state university, which will be referred to as State University throughout this paper, mirrors the problems reported by other sources. Although females currently comprise more than half of the university's student population, only 26% of the CIS students are women. (As students of Caucasian heritage comprise the great majority of the population of this university, the picture relative to minority enrollment at State University is less clear.) Moreover, enrollment in the CIS major and/or minor at this university, reported at 381 in 1986-1987, fell to 294 by 1990-1991 and continued to drop through the early 1990s. Attrition in the programming courses is high, and students, especially those from other disciplines taking the initial programming class as a service course, are often less than successful. Unfortunately, the Deimel group (1983) observation that "drop out rates are high; students feel overworked; and many of them, even those who eventually pass, seem perpetually 'lost' in their first course" (p.12) seems all too often true.

Thus, it would seem there is reason to be concerned about the current enrollment patterns in the computing discipline. Technological skills will become increasingly important as our country enters the 21st century. Lucrative professional opportunities in information systems await qualified applicants. In light of the high percentage of White males currently employed in the field and national concern with affirmative action, businesses continue to seek out qualified women and minorities. In spite of acknowledged career opportunities and known financial advantages, interest in the computing sciences continues to decline. It is important, therefore, to determine what factors are driving students away.

Many students find college-level computer programming courses demanding, and attrition rates in those courses are high. Lamenting a high mortality rate of subjects in a study of computer programming students at the University of Saskatchewan, Greer (1986) described the problem as "an unfortunate but inevitable consequence of the nature of university introductory computer science courses" (p. 219). Hostetler (1983), speaking of programming students at the University of Illinois at Urbana-Champaign (an institution with high admissions criteria) also noted the broad range of student programming abilities. Others (Deimal, Hodges, & Moffat, 1983) have observed students' "invariable lack of intuition" (p. 11) about the nature of programming.

Various authors offer reasons why students find programming courses difficult. Pea and Kurland (1985) assert that programming requires the programmer to transform an ill-defined problem into a well defined one and then solve it. Perkins, Schwartz, and Simmons (1988) first dismiss the notion that programming is not hard to learn, but merely poorly taught. They then propose that programming is both problem-solving intensive and precision intensive. As an example, Perkins et al. contrast a 90% correct program with a 90% spelling test. The spelling test, they say, likely earns the student an 'A'. On the other hand, they suggest that a 90% correctly coded program will not produce an acceptable outcome, will probably make the debugging (finding and correcting errors) task extremely demanding and frustrating, and will almost certainly not earn an 'A'. Deimel et al. (1983) and Joni and Soloway (1986) remind programming instructors that not only do teachers expect programs to work, they expect them to be robust, user-friendly, well-documented, modifiable, and so on: an undeniably difficult task. Hancock (1988) summarizes the problem with the observation that teaching and learning computer programming at the introductory level are difficult, that many students have had unrewarding experiences and come away believing " 'I'm not the kind of person who can do this' " (p. 18).

Factors of Success

In light of the preceding observations, it is not surprising that the literature of the field features numerous reports of studies that attempt to identify factors of computer programming success. Mathematics-oriented factors have been the traditional fare of these articles (See Alspaugh, 1972; Chung, 1988; Koubek, LeBold, & Salvendy, 1985; Kwan, Trauth, & Driehaus, 1985; Leeper & Silver, 1982; Mandinach & Linn, 1986; Oman, 1986; Sharma, 1986-87; Stephens, Wileman, & Konvalina, 1981). Many of these authors found a positive correlation between mathematics factors and programming success, but others challenged the finding. Numerous other studies have investigated the relationship of demographic variables, analogical reasoning ability, spatial ability, college grade-point average, age, gender, high school performance, verbal ability, prior programming experience, scores on IBM's Programmer Aptitude Test, and semester in school to programming success (Bauer, Mehrens, & Vinsonhaler, 1968; Clement, Kurland, Mawby, & Pea, 1986; Fowler & Glorfeld, 1981; Greer, 1986; Hostetler, 1983; Johnson & Johnson, 1991; Kagan, 1988; Mazlak, 1980; Newstead, 1975; Sharma, 1986-87; Wileman, Konvalina, & Stephens, 1981).

Other factors may determine a student's likelihood of success with computer programming. People classified as Introvert-Sensing-Thinking type (students who tend to concentrate quietly, pay attention to details, have a talent for memorizing facts, and demonstrate perseverance) as defined by the Meyers-Briggs Type Indicator tend to do better in computer science courses than other personality types (Sharma, 1986-87). In addition Stevens (1983) and Cavaiani (1989) found that people with a field independent cognitive style were more successful than those who were field dependent. Johnson and Johnson (1991) also found positive correlations with neuroticism and introversion.

Although some of the aforementioned studies did find positive correlations between the named factors and computer programming ability, the studies generally feature some shortcomings. First, most of the studies are atheoretical. Sheil (1981), commenting on the atheoretical nature and weak methodology of much computer programming research, noted the extreme difficulty inherent in designing a meaningful experimental study of such a complex cognitive skill as computer programming.

Next, the studies attempted to correlate programming success with programmers' background variables. While these factors may assist advisors counseling students regarding the selection of programming classes, they do not help the classroom teacher very much. Teachers cannot alter their students' age, gender, verbal ability, high school rank, cognitive style, or other background data. Howell (1993), who was mainly concerned with women's persistence in an undergraduate computer science major, noted the limitations of background-related studies, and called for studies that would describe the factors that shape the culture of computer science classes and individual students' responses to those factors.

Finally, the quantitative studies do not attempt to explain the students' conceptions or misconceptions. From a study that related computer programming success with such cognitive characteristics as spatial ability, field independence, visualization ability, logical reasoning, and direction following, Foreman (1990) concluded, "One clear instructional implication is that alternate modes of instruction should be available" (p. 60). Teachers need research that informs the instructional and curricular decisions over which they have control. As Spohrer and Soloway (1986) report, "The more we know about what students know, the better we can teach them" (p. 624).

Cognitive Style

Researchers (Cheny, 1980; Johnson & Johnson, 1991) investigating personality variables regularly find positive correlations between programming ability and people with an analytic cognitive style. Those with an analytic style approach decision-making in a structured manner, try to reduce problems to a set of underlying causal relationships, build models, choose optimal alternatives, and do mathematical analysis. People with a heuristic style, on the other hand, emphasize common sense, intuition, trial-and-error, ad hoc sensitivity analysis, "muddling through," use of feedback to adjust the course of action, and satificing, that is, choosing the first reasonable alternative that presents itself (Cheny, 1980).

Hence, whether the attribute is called formal reasoning, or hypothetico-deductive thinking (Lawson, 1985), or working memory capacity, or cognitive style, the literature documents the notion that some groups of adults will find themselves more comfortable than others with computer programming. Furthermore, the differences cannot be dismissed on the basis of intelligence alone.

In contrast, other research (Turkle & Papert, 1990) has documented the programming success of students with a concrete or heuristic learning style. Using the term "planner" to refer to people for whom the top-down, structured, analytical approach is natural, Turkle and Papert classify people who think concretely as "bricoleurs." Bricoleurs, the authors say, construct "theories by arranging and rearranging, negotiating and re-negotiating with a set of well-known materials" (p. 136). According to Turkle and Papert, bricoleurs prefer a transparent style of program construction, staying in touch with the details of the program at all times. Planners, on the other hand, prefer an opaque "black-box" approach, in which procedural abstraction hides the program details.

Turkle and Papert (1990), who agree with Piaget's view of the "diverse nature of different kinds of knowledge" (p. 129), disagree with Piaget's description of the diverse forms as stages, viewing the "different approaches to knowledge as styles, each equally valid on its own terms" (p. 129). These authors suggest that people who approach programming in a nontraditional (concrete) way are discouraged by the dominant computer culture. These students are not computer phobic; they do not stay away because of fear or panic. The way of thinking perpetuated by the formal, rule-driven, hierarchical approach to computer programming, however, forces them to interact in a fashion that is incompatible with their learning style. Rules do not exclude them from the computer culture; the way of thinking makes them reluctant to join.

Thus, it becomes difficult to conclude *a priori* that computer programming per se is innately foreign to some people. Were such a conclusion the case, the low enrollment of females and minorities and the difficulties students experience might be considered beyond pedagogical control. The goal instead must be to discover what the novice programmer's conceptions are and in what ways, if any, the conceptions of those with a contextual or concrete cognitive style differ from those with a more analytic style. In this way, it may become possible to create a computer programming classroom climate that is less hostile to the contextual thinker.

Technical Information

This section of the chapter introduces technical information that is central to the study reported in this paper. A brief example program follows a discussion of structured programming, and pertinent technical terms are defined in the glossary that concludes the technical information section.

Structured Programming

Background. When investigating student programming activities, it is important to begin with "structured programming." The structured programming paradigm emerged during the 1960s as a way of reducing program complexity. Prior to that time, programs were developed using code that lacked structure, a programming style known as "spaghetti code." Dramatic hardware advances were making it possible to automate increasingly sophisticated applications, and commercially developed spaghetti-code programs were increasingly being placed in production with undiscovered errors. In 1964, mathematicians Guiseppe Jacopini and Corrado Bohm proved that any program logic could be expressed using only three control structures: sequence, iteration, and selection (Nance & Naps, 1989).

The first time structured programming methods were applied to a large scale application (IBM Corporation's New York Times Project) resulted in greatly increased programmer productivity and a phenomenally low error rate (Nance & Naps, 1989). Faced with evidence of success and due largely to the efforts of Edger Djikstra, the professional programming community embraced the structured programming paradigm. Since then, students enrolled in information systems or computer science courses have been taught using a structured programming language. Pascal, a programming language developed by Nicklaus Wirth specifically to teach structured programming, is one such language.

Modularity. In addition to incorporating the three aforementioned program control structures, structured programming languages must support modularization. Nance and Naps (1989), for example, define structured programming as "the process of developing a program where emphasis is placed on the correspondence between independent modules" (p. 269). They continue:

> Structured programming is especially suitable to large programs being worked by teams. By carefully designing the modules and specifying what information is to be received by and returned from the module, a team of programmers can independently develop their module and then have it connect to the complete program. (p. 269)

A module is a collection of programming statements which work together to perform a single logical function, and modularity is the first principle of structured programming (McIntyre, 1991, p.62). Modules are linked together to form programs; the connections or links between these modules are specified in parameter lists (McIntyre, 1991; Nance & Naps, 1989).

Variables. An introduction to variables at this point is fundamental to any discussion of parameters. Leestma and Nyhoff (1993) liken variables to mailboxes. Their glossary defines a variable as "an identifier associated with a particular memory location. The value stored at this location is the value of the variable, and this value may be changed during program execution" (p. 967). Within a program, exactly one memory location corresponds to a variable, and a variable can hold only one value at a time.

McIntyre (1991) points out that this is only reasonable; it would be impossible for a programmer to keep track of variables and their values otherwise.

McIntyre (1991, p. 27) provides a pedagogic view of the term variable. He first suggests that variables are the single most difficult concept for many naive programmers. Distinguishing between the terms changing and changeable, McIntyre proposes that a changeable value is one that is determined rather than arbitrary. In common usage, he reports, people employ the term "variable" to stand for something that changes, often in an uncontrolled or unpredictable way. He offers that if most people are asked to give an example of a variable, they might respond "the weather." This, he says, is not a useful concept in programming, where the value of a variable is changed in a controlled manner. McIntyre further posits that beginners have difficulty distinguishing between the name of a variable, which is constant, and a variable's value, which is changeable.

Invocation. In addition to modularity and variables, a third important concept in structured programming is "invocation." In a modular program, control of the computer is passed to a module (Pascal procedure) when the module is invoked or called. The module retains control until its task is completed; then control is returned to the original or calling module (McIntyre, 1991). Invocation involves passing information between modules, and parameters are the mechanism by which information is passed between program modules (Leestma & Nyhoff, 1993; Mercer, 1992). One great benefit of parameter (argument) passing is that it allows different modules to refer to the same variable [memory location] with different names (McIntyre, 1991).

Significance of parameters. BASIC, the programming language often taught to pre-college students, typically does not support parameter passing. In other words, all of its variables are "global" (requiring all modules to refer to a variable by the same name). For large programs developed by teams of programmers, the requirement is unreasonable. McIntyre (1991) offers that it is this feature, more than any other, which limits BASIC's usefulness as a programming language, claiming, "Of all the reasons for BASIC not being taken seriously as a programming language, the fact that most versions are restricted to global variables is the most compelling" (p.46). Contrasting the power of Pascal with the limitation of BASIC, the author continues:

> Pascal, on the other hand, is typically written so that all variables are local, and their names are restricted to the particular module in which they are defined. Because of this, each module or subroutine can be viewed as an independent unit. Values for variables can be passed as parameters for the modules or subroutines, and they need not be identified by the same variable name. When necessary, Pascal does allow for the assignment of values to global variables. Local and global variables are an integral part of Pascal, and they are one of the compelling reasons for Pascal being a favorite language for proponents of structured programming. (p.46)

Parameter definitions. The Cooper and Clancy (1985) glossary defines parameter as follows:

> In general, a particular value that is substituted for a general term. In Pascal, a parameter is a variable created in the *parameter list* portion of a subprogram: a *value-parameter* is a local variable whose starting value is *passed* as an *argument* to the subprogram, and a *variable-parameter* is a local re-naming of a (relatively) global variable. These are also called *formal parameters*. (p. 572)

In the body of their book, the authors of the well-known text note that changing a value parameter has no effect on its actual parameter (argument). Conversely, "changing the variable parameter *does* affect its argument, since they're exactly the same variable" (p. 72). The Cooper and Clancy definitions of value and variable parameters provide the basis for what this paper will refer to as the *copy versus the shared memory-location view of parameter passing.*

In contrast, Leestma and Nyhoff (1993), authors of the text used by the students in the current study, define a value parameter as a "formal parameter of a subprogram that is used to pass information to the subprogram but not to return information from it" (p. 967) and variable parameter as "a formal parameter of a subprogram that is used both to pass information to, and return information from, the subprogram" (p. 967). The Leestma and Nyhoff definitions provide the basis for what this paper will refer to as the *one-way versus two-way communication view of parameter passing.*

Taxonomy of parameter-using programs. According to Derek Nazareth (personal communication, September 1994), programs may be categorized according to the following taxonomy in terms of parameter passing:

Programs without modules; they hence have no parameter passing.
Modular programs that use global variables; they hence have no parameter passing.
Modular programs that pass simple parameters.
Modular programs that pass structures composed of the same type of elements (arrays).
Modular programs that pass structures composed of different types of elements (records).

Modular programs that pass pointers to simple data elements.

Modular programs that pass pointers to structures.

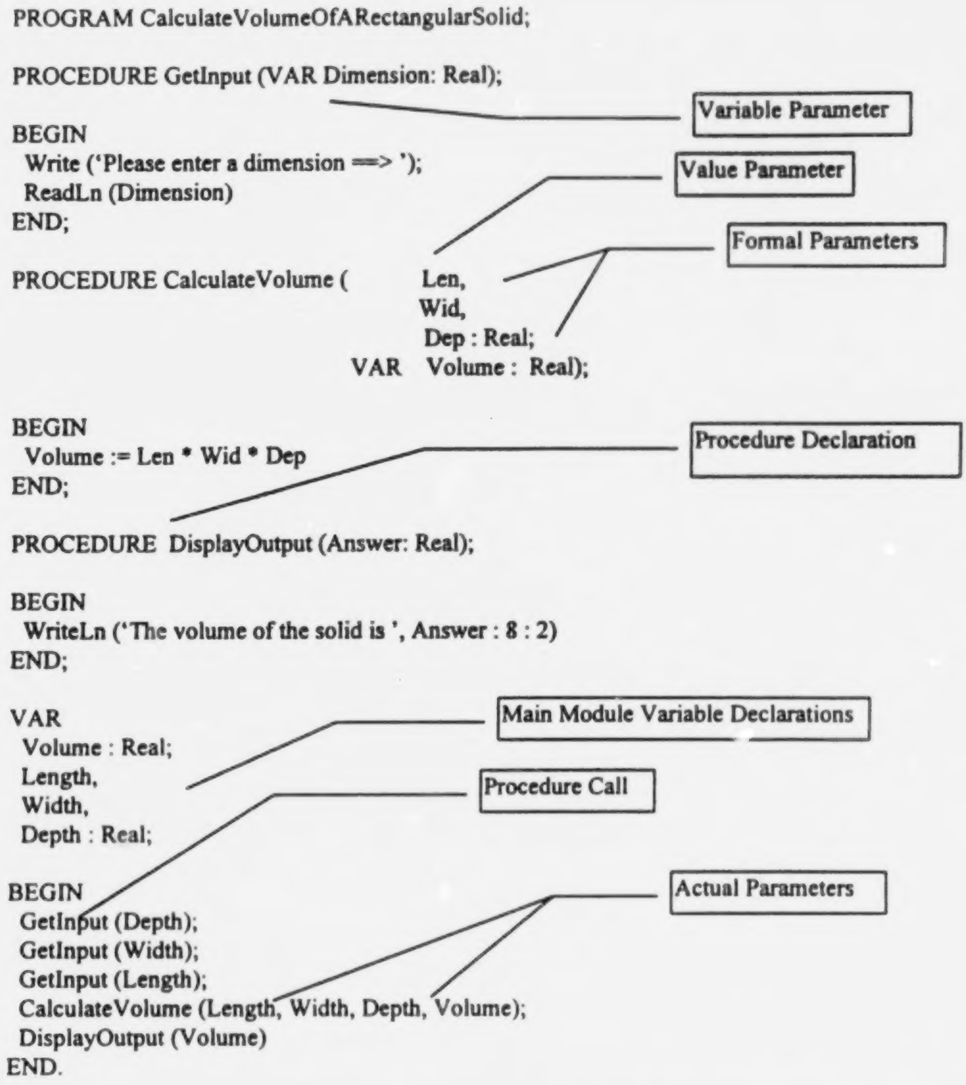
Modular programs that pass control data.

The study described later in this paper reports on students learning to program using simple parameters. The decision to study that category of programs was made because programs at the simple end of the continuum (those with no parameters) are useless code in all but the most trivial of circumstances. On the other hand, however, students must learn to use simple parameters before they can use more sophisticated techniques.

Summary. Structured programming requires modules, variables, and parameters. Without parameters, there is no real programming, yet the literature reveals a virtual absence of attention to the construct of parameter passing. It is the view of some programming instructors that no single construct poses greater problems for the novice programmer. For example, James Gifford, a professor of computing at the university in question, is one such instructor. When asked about the introductory programming course he regularly teaches, he said, "The heart of the course is the notion of parameters. Conceptually, it's the hardest thing for the students to get" (personal communication, October 1994). Thomas Naps, who developed an algorithm visualization program especially designed to assist novice programmers to understand computer science concepts, concurred (personal communication, August 1991).

Pascal Example

A brief example may help clarify terminology. The example, shown in Figure 1, takes advantage of the Turbo Pascal extension, which allows the module declarations to precede the main module variable declarations.



```
PROGRAM CalculateVolumeOfARectangularSolid;

PROCEDURE GetInput (VAR Dimension: Real);

BEGIN
  Write ('Please enter a dimension ==> ');
  ReadLn (Dimension)
END;

PROCEDURE CalculateVolume (        Len,
                                  Wid,
                                   Dep : Real;
                             VAR   Volume :  Real);

BEGIN
  Volume := Len * Wid * Dep
END;

PROCEDURE  DisplayOutput (Answer: Real);

BEGIN
  WriteLn ('The volume of the solid is ', Answer : 8 : 2)
END;

VAR
  Volume : Real;
  Length,
  Width,
  Depth : Real;

BEGIN
  GetInput (Depth);
  GetInput (Width);
  GetInput (Length);
  CalculateVolume (Length, Width, Depth, Volume);
  DisplayOutput (Volume)
END.
```

Variable Parameter
Value Parameter
Formal Parameters
Procedure Declaration
Main Module Variable Declarations
Procedure Call
Actual Parameters

Figure 1: Modular Pascal program that uses simple parameters.

BEST COPY AVAILABLE

Glossary of Technical Terms

Unless otherwise noted, the following definitions are taken from Leestma and Nyhoff (1993), the textbook used by the students reported in this study.

Term	Definition
Actual Parameter	A parameter that appears in a procedure/function reference and is associated with the corresponding formal parameter in the procedure/function heading (p. 957).
Formal Parameter	A parameter used in a subprogram heading to transfer information to or from the subprogram (p. 961).
Global Variable	A variable that is known throughout the program (Horn, 1995, p. 127.
Iteration (repetition) Structure	A control structure in which one or more statements are repeatedly executed (p. 965).
Local Variable	A variable whose scope is a subprogram (p. 962).
Modular Programming	A programming strategy in which major tasks to be performed are identified and individual procedures/functions (called modules) for the tasks are designed and executed (p. 963).
Parameter	A variable that is used to pass information between program units (p. 964).

Term	Definition
Procedure Reference (Call, Invocation)	A Pascal statement that is used to call a procedure (p. 964). Activates the named procedure. Execution of the current program unit is suspended, and execution of the procedure begins. When the end of the procedure is reached, execution of the original program unit resumes with the statement following the procedure statement (p .177).
Selection Structure	A control structure in which one of a number of alternative actions is selected (p. 966).
Sequential Structure	A control structure in which the statements comprising the structure are executed in the order they occur (p. 966).
Structured Program	A program that is designed using only the three basic control structures: sequential, selection, and repetition (p. 966).

Term	Definition
Value Parameter	A local variable, used only in the subprogram, whose starting value is given by an argument in the subprogram call. Changing the value parameter has no effect on its argument (Cooper & Clancy, 1985, p. 72). Formal parameter of a subprogram that is used to pass information to the subprogram but not return information from it (p. 967).
Variable	An identifier associated with a particular memory location. The value stored in the location is the value of the variable, and this value may be changed during program execution (p. 967).
Variable Parameter	An alternate name, meaningful only in the subprogram, for the variable that's supplied as its argument in the subprogram call. Changing the variable parameter does affect its argument, since they're exactly the same variabl (Cooper & Clancy, 1985, p. 72). A formal parameter of a subprogram that is used both to pass information to, and return information from, the subprogram (p. 967).

Purpose of the Study

The purpose of this study is to describe a novice programmer's understanding of the construct of simple parameter passing, with the ultimate goal of improving the teaching of the construct. The study will specifically address the following questions.

1. What action knowledge does the novice programmer bring to the study of parameter passing?
2. What intuitive conceptions does the novice programmer possess regarding parameter passing?
3. How does the novice express principled understanding of the parameter passing construct?
4. What is the relationship between a novice's conceptual and procedural knowledge of parameter passing?
5. What impact does a student's understanding of parameter passing have on the eventual success in an introductory programming course?
6. What impact does a student's understanding of parameter passing have on his or her disposition towards computer programming?

The process of learning and understanding parameter passing is built on one's ability to construct personal realities. Thus it is necessary to look at currently accepted theories of how learners construct knowledge. Those theories introduce the second chapter.

CHAPTER II

LITERATURE REVIEW

Framework: Constructivism

Constructivism contrasts the idea of socially constructed meanings with the objectivist view that meaning exists separately from the act of interpretation. Knowledge, to the constructivist, is not an "inert, acquirable commodity" (Sack, Soloway, & Weingrad, 1993, p. 387) that exists independently, waiting (in a lecture or a text for example) to be discovered or communicated. Constructivists believe that people, rather than being empty vessels, fashion their own meanings for the phenomena they encounter. Learners, using the sum total of their life experiences, construct their own senses of reality. Learning proceeds from the inside out; seeds in the form of percepts are acquired and cultivated from within. Learners select and interpret new information in terms of what they already know (their existing schemas), and they change and reorganize their existing knowledge (schemas) in terms of the new knowledge (Gagné, Yekovich, & Yekovich, 1993; Linn, Sloane, & Clancy, 1987; Neuman, 1993; Tuovinen & Hill, 1992).

A constructivist view underscores the social nature of learning. Bruner (1973) cautions, "The psychologist or educator who formulates pedagogical theory without regard to the political, economic, and social setting of the educational process courts triviality and merits being ignored in the community and in the classroom" (p. 115). Citing John Seely Brown, Sack et al. offer that "learning is at core a process of enculturation, of entry into a culture or community of practice" (1993, p. 387). The authors suggest that until recently, problem solving has focused on individuals "de-contextualized" from their social fabric. In these authors' view, those involved in activities of knowledge production (science, for example) and knowledge reproduction (education, for example) are concerned with recruiting, persuading, and enculturating others. According to this theory, learning involves moving into a community or culture of practice.

Constructivism is based on the interpretivist philosophy of social reality as mind-dependent and symbolically constructed. In this philosophical view, reality is socially and historically conditioned and value laden. There is no single point of view; reality depends on the perceptions, views, and purposes held in community with others. Truth is a matter of credibility; it is what those involved agree it is. Valid is a label applied to an interpretation with which one agrees (Hatch, 1985; Smith & Heshusius, 1986).

Arguing for the interpretivist philosophy, Hatch (1985) observed that whether the universe is ordered can neither be proven nor disproven. Regularity and statistically significant predictability, according to Hatch, neither explain events nor prove that events will always occur. The interpretivist seeks to understand how people define reality, for it is that definition that governs people's behavior and their interaction with the universe.

Associated with the interpretivist or phenomenological philosophy is the qualitative research paradigm. The strength of qualitative research resides in its potential to generate rich, detailed process data. Typically, qualitative research is inductive, process and discovery oriented, uses subjective measurement, employs the inquirer as the instrument, is non-generalizable, and reports the insider's perspective. Qualitative research seeks not to predict, but to understand, and to find meaning in the social phenomena being examined. This study adopted a constructivist perspective and employed the qualitative research methods associated with the view.

Theoretical Bases of the Study

The study reported in this paper draws on a variety of theories. The work of cognitive psychologist John Anderson is included, as it often frames computer programming research. Because of the abstract nature of the computer programming activity, Jean Piaget's work is central. Lev Vygotsky's social interactionist theory provides a foundation for the cognitive apprenticeship approach that is currently receiving much attention in the computer science education community. The mental models literature, combined with the cognitive-style research reported in the first chapter, furnishes a basis for understanding the varied ways learners process information. Finally, Linn and Songer (1991) supply a model for conceptual change that seems especially to pertain to novices learning to program.

Cognitive Psychology: Anderson

Although cognitive psychologist John Anderson's (1983) ACT* (Adaptive Control of Thought) theory might be considered more positivist than constructivist, Lawson et al. (1991) use it to frame their constructivist hypothesis. Irrespective of the philosophy that it embodies, Anderson's theory is important for this study because it frames so much of the computer programming literature.

Anderson theorizes that information is received by sensory receptors and registered in the central nervous system in immediate memory, where information is held for a very brief time. Then, by a process of selective perception, a limited amount of information is transferred from immediate to working memory. Working memory, which in humans is extremely limited, corresponds to awareness. Information that is not lost is transferred from working memory to long-term memory, which in humans is virtually inexhaustible. Long term memory is divided into declarative memory and procedural memory. Procedural knowledge is knowing how to do something (skill knowledge) while declarative knowledge is knowing about something, for example conceptual or factual knowledge (Gagné et al., 1993).

Skills, which are stored as production or IF . . . THEN, systems (IF *declarative knowledge* THEN *action*) begin as declarative knowledge. In early stages of skill acquisition, skill behaviors are "interpreted" one step at a time, making large demands on working memory. As skills are practiced, the steps become larger, and the demands on working memory lessen. Finally, after much practice, some skills (primarily basic skills) become automated (compiled) and are performed without conscious thought (Gagné et al., 1993).

Complex skill (computer programming, for example) is composed of declarative or conceptual knowledge, automated basic skills, and strategies or non-automated skills

(Gagné et al., 1993). According to Anderson's theory, practice is a crucial component of knowledge compilation. Thus, ACT* is seen as a theory of learning by doing. The practice component of the theory is very appealing, since the programming community universally attests to the notion that learning to program requires many hours of practice.

Piagetian Theory

Piaget's constructivism. Piaget theorized that intellectual functioning is characterized by the complementary processes of adaptation and organization. Adaptation, the mind's ability to alter the organization of the structures or schemas, is composed of two subprocesses, assimilation and accommodation. Assimilation is the process whereby new knowledge is interpreted in terms of current understandings (schemas or structures). If the new knowledge provokes conflicts, contradictions, or incompatibilities, a state of disequilibrium is invoked. In order to return to equilibrium, current understandings (schemas or structures) are transformed to accommodate the new understanding and bring about a return to equilibrium. Organization is a self-regulating mechanism that provides for a balance between assimilation and accommodation, and makes adaptation possible. Learning is thus the continuous process of active assimilation and accommodation, according to Beilin (1992) and Neuman (1993).

To restate, individuals need to form their own hypotheses and keep trying them out through mental actions and physical manipulations: observing what happens, comparing their findings, asking questions, and discovering answers. When objects and events resist the working model an individual has mentally constructed, he or she is forced to adjust the mental model to account for the new information, reject the new information, or modify it artificially. The mental models (schemas) are continually reshaped, expanded, and reorganized by new experiences.

Piaget's stage theory. Piaget's stage theory, at least a later version, is also important to any discussion of the cognitive demands of computer programming. As widely documented, Piaget first theorized that all children pass through certain predictable stages culminating in the stage of formal operations. Certain characteristics of the formal operations stage are analogous to the skills believed necessary for computer programming (Bastian et al., 1973; Cafolla, 1986-87; Dalbey & Linn, 1985; Fischer, 1986; Kurtz, 1980; Madison & Gau, 1993; Nachmias, Mioduser, & Chen, 1986). Faced with mounting empirical evidence contradicting the theorized universal distribution of formal thinking in adults (Lawson, 1985; Mitchell & Lawson, 1988), Piaget later acknowledged that "formal operations may not be attained by all people in all areas of judgment" (cited in Shaklee, 1979, p. 339). Lawson states unequivocally, "Many people never become formal thinkers. This seems crystal clear" (p. 591).

Fischer (1986) elaborates on components of an introductory Pascal course that require formal thought. Central to this paper is the attention the author devotes to the concept of top-down design (of which parameters are a crucial aspect). Fischer quotes from Dale and Orshalik, authors of a frequently used Pascal textbook:

> We start by breaking down the problem into a set of sub-problems. This process continues until each sub-problem cannot be further divided. We are creating a hierarchical structure, also known as a tree structure, of problems and sub-problems called functional modules. (p. 11)

In order to create such a hierarchy, Fischer contends, the programmer must conceive of all possible steps and the order in which they occur. Once the student can no longer rely on concrete experience with a problem, breaking the program down into component parts requires formal thought and an understanding of interrelated, but unstated components. Although Fischer fails to mention parameters, it is important to remember that that parameters are the communication links that connect the functional modules.

Vygotsky-related Theory

Zone of proximal development. According to Wertsch and Tulviste (1992), Vygotsky posited that all higher order mental functions begin in social interaction and are only later internalized psychologically. Any function, he theorized, appears in a child's development twice, first socially and then individually. In conjunction with this theory, Vygotsky is widely known for his concept of the zone of proximal development (ZPD). Measured in mental age equivalents, the ZPD is the difference between a child's potential and actual development levels in any given domain-specific context. The upper limit of the zone (potential development level) is the child's problem-solving ability given the assistance of an adult or more able peer. The lower limit is the level at which the child can function independently (actual development level). In this view, an individual's potential development level becomes more important than the actual. Learning is seen as an interpersonal, dynamic event depending on two minds, one better informed or more skilled than other. Learners appropriate for themselves modes of thinking that they observe in an expert (or more able peer), while the duo jointly solves a problem.

Linn and Songer (1991) define the ZPD as the "range of all possible cognitive changes that might result for students with a given set of initial understandings and particular learning environment" (p. 382). Effective instruction enables conceptual change, and the ZPD acts as a guide to determine the point at which instruction would be the most effective. From this perspective, the only good teaching is that which outpaces actual development and leads it through the ZPD (Simon, 1987; Tharp & Gallimore, 1989).

Emulating the natural teaching of home and community and the everyday interactions of domestic life, the teacher's role becomes one of supporting or guiding, and facilitating learning and development, as opposed to the transmitting of knowledge. Based on the assumption that learners can first do things in a supportive context and later independently in a variety of contexts, the teacher provides a "scaffold," assistance that enables the learner to move to the next level of independent functioning (Lemerise, 1993). The mechanism for passing cognitive skill from teachers to pupils is the transfer of responsibility for reaching a current goal, whatever the goal might be. The venue for transfer is ZPD (Belmont, 1989).

Cognitive apprenticeship. Embodying a Vygotskian view of learning, cognitive apprenticeship is an instructional paradigm based on the time-honored tradition of novices learning a trade by working closely with the masters (Collins, Brown, & Holum, 1991). Traditional apprenticeship is comprised of four important aspects: modeling, scaffolding, fading, and coaching. In modeling, the master demonstrates the different parts of the task, making the processes visible for the student. The master provides scaffolding or support for the students as they carry out the task, and gradually fades or removes the support as the novices become more adept and able to assume responsibility. The coach at this stage oversees the apprentice's tasks, provides hints, evaluates activities, provides challenges, offers encouragement, and gives feedback. The interaction of scaffolding, observation, and increasingly independent practice assists the apprentice in developing correction and self-monitoring skills, and in integrating conceptual knowledge with skill (Collins et al., 1991).

Collins et al. (1991) contrast traditional apprenticeship with cognitive apprenticeship. The former, they note, usually involves tangible, physical activity in which the process is readily observable. In the latter, however, thinking must be made "visible," and special attention must be given to the methods experts depend on to acquire or use knowledge in addressing real-life problems.

Modeling, coaching, and scaffolding are the core teaching methods advocated by Collins et al. (1991). Teachers, they claim, should supplement the core teaching methods by evoking the students' articulation of their knowledge or problem-solving strategies. Encouraging students to reflect on and compare their own problem-solving processes with those of others, and encouraging students to explore on their own can promote this process. Tasks, the authors propose, must be sequenced so that global skills are stressed first (enabling students to develop a conceptual model), and should increase in complexity and diversity.

Lastly, Collins et al. (1991) address the sociology of the learning environment. Crucial to their model is the element of situated learning wherein students must understand the purpose of what they are learning. They must construct knowledge actively rather than receive it passively, and they must learn the different conditions under which their knowledge can be applied. Common projects and shared experiences of students working together in communities of practice, the authors contend, lead to a "sense of ownership, characterized by personal investment and mutual dependency" (p. 45).

Mental Models

Mayer (1985) described a user's mental or conceptual model of a computer programming system as the "conception of the information-processing transformations and states that occur between input and output" (p. 90). Observing that mental models vary in their usefulness, he asserted that more useful models assist sophisticated programming performance. The models also vary in their veridicality, with the more veridical models matching the operations of the machine's language more closely. Mayer described a good model as one "that helps novices to successfully perform on programming problems" (1985, p. 91), and referred to Du Boulay's requirement that good mental models should be transparent (the internal operations and states should be visible to the learner).

Comparing the naive conceptions of beginning programmers to the naive conceptions of students learning science, Mayer (1985) theorized that learning a programming language involves acquiring several types of knowledge. A beginning programmer gains knowledge of syntax, learns how to generate a program, and develops a mental model of the computer system. Reporting on research with BASIC students, Mayer offered that students with less mathematical experience or lower mathematical abilities seemed to profit most from direct instruction on a mental model early in their programming experience. Moreover, high-ability subjects in Mayer's study did not appear to be hindered by the mental model instruction.

Perkins et al. (1988) described BASIC programmers' knowledge as fragile, speculating that programming behaviors could not be accounted for simply in terms of missing knowledge. Basing their findings on such examples as plausible but garbled commands, the Perkins group offered that students failed to hand-execute the BASIC commands "with a mental model of the machine in mind" (p. 157), either because the students lacked a model or neglected to evoke such a model. Attributing the fragile programming knowledge to the students' lack of a mental model, these authors contended that a mental model of the computer would allow students to check tentative responses found in a fragile knowledge base. Citing Mayer's work, the Perkins group advocated using a visual paper computer, a functional model of the BASIC language.

Norman (1983), in mild contrast to other authors, distinguished between conceptual and mental models. Conceptual models, he claimed, are developed as tools to help understand physical systems; mental models are what actually exist in people's minds to guide their use of the physical systems. Ideally, conceptual and mental models should have a direct relationship. Norman concluded that an effective mental model enables the user to predict the behavior of a target system.

Du Boulay (1986) proposed that even if no effort is made to give novices a view of the inner workings of a computer, they develop one of their own: a view that may be impoverished. He wrote that novices may rely on coincidences, and that their models are often insufficient to explain observed behavior. A desirable conceptual model, he submitted, has two important characteristics: simplicity and visibility (1989). Moreover, the notional (or virtual) machine must be built upon the characteristics of the language. In other words, a BASIC machine would not be the same as a Pascal machine. Furthermore, the model should be a "glass box," emphasizing the processes that go on inside the machine, rather than a "black box" that emphasizes inputs, outputs, and a process without concern for what actually happens (Du Boulay, cited in McIntyre, 1991).

According to McGrath (1990), mental models may take the forms of analogies, images, or metaphors. Analogies, she proposed, help students understand the relations among actions. Observing that both novices and experts employ mental models, McGrath counseled teachers to think carefully about the models they select. The well-chosen model must be simple enough for students to comprehend and employ, but should be equally useful to the expert. McGrath noted that good mental models are most helpful for the developing students, and concluded that witnessing the students' understanding is the reward for incorporating carefully selected models. McIntyre (1991), in his textbook on teaching structured programming, agreed that "the use of models is one of the most powerful tools of science and technology" (p. 161).

Mayer (1989), reporting on his use of a concrete model of a BASIC computer--memory scoreboard, input window, arrow pointing to a program list, and an output pad--summarized the position on mental models with the statement:

> These results provide clear and consistent evidence that a concrete model can have a strong effect on the encoding and use of new technical information by novices. These results provide empirical support to the claims that allowing novices to "see the works" allows them to encode in a more coherent and useful way. When appropriate models are used, the learner seems to be able to assimilate each new statement to his or her image of the computer system. Thus, one straightforward implication is: If the goal is to produce learners who will not need to use the language creatively, then no model is needed. If the goal is to produce learners who will be able to come up with creative solutions to novel (for them) problems, then a concrete model early in learning is quite useful. (p.147)

Linn and Songer Model for Conceptual Change

The three stages established by Linn and Songer (1991) in their model for conceptual change include action knowledge, intuitive conceptions, and scientific principles. Built on a constructivist philosophy, the model focuses on the understandings students bring to a learning situation, their initial conceptions, and the mechanisms that decide the rate of the constructive process (Linn & Songer, 1991). The authors proposed that some cognitive change theories are domain-specific while others are domain-general. Some theories posit revolutionary conceptual changes, and still other theories claim changes are evolutionary. Linn and Songer proffered an integrated perspective, advising researchers to acknowledge that all viewpoints contribute to understanding cognitive change and concept development. Posing dichotomies is unproductive for analyzing complex situations, they claiamed. Since computer programming is unquestionably a complex, cognitively demanding situation, this study attempted to follow their lead.

Originally posed for students learning science, the Linn and Songer (1991) three-stage model holds great appeal as a framework for describing novice programmers' acquisition of Pascal programming constructs. First, it allows for domain-independent developmental constraints outlined by the Piagetian-related theories. However, the model also allows for domain-specific, plan-oriented theories that dominate computer programming literature. The model allows for evolutionary changes such as increased working memory capacity or progress through the ZPD, but still acknowledges the "a-ha!" Certainly students acquire a gradually increasing repertoire of programming templates and skills, but they also experience revolutionary changes (Linn, 1992). In fact, the novice in the pilot study reported in this paper, after struggling extensively to understand parameter passing, described success as a "mountain-top experience."

Action knowledge. During the first stage of the model (Linn & Songer, 1991), learners acquire action knowledge. Reminiscent of Piaget's theory of how young children learn, these authors theorized that learners acquire action knowledge from action, observation, and unreflective responses to events they encounter. Their action knowledge summarizes their actual experience.

Although in reality all computer science concepts do find their bases in natural world occurrences (Nance & Naps, 1989), the gulf between the abstract computer science concepts and natural world experiences the concepts imitate is wide. Students rarely learn the concepts using unguided discovery, perhaps because the gap between concrete life experiences and the abstract programming constructs is just too great to be bridged without instruction designed to make explicit the connections.

Intuitive conceptions. In the second stage of the model, learners combine action knowledge into intuitive conceptions, conjectures they make to explain events they observe. Using a process of reflective abstraction (wherein people combine information available to them by reflecting on inconsistencies and commonalties), learners ground their intuitive conceptions in experience and intentionally organize knowledge to make sense of the world and reduce its complexity (Linn & Songer, 1991). The Linn and Songer position is in accord with Neuman's (1993) definition of learning, which asserts that students learn when they realize

> that knowledge they have acquired either intuitively or from experience inadequately or inaccurately explains observed phenomenon. They recognize the incongruity or inconsistency between their expectations and the firsthand evidence. They then actively construct a new reality that is personally understandable and plausible and that fits the new evidence. (p. 94)

However, because students are more likely to base their ideas solely on personal experience, unguided discoveries often lead to flaws or conceptions that differ from those of scientists. Although these flaws are generally labeled misconceptions, Linn and

Songer prefer the term "intuitive conceptions," alleging that it emphasizes the positive nature of a student's constructive process.

Although research on the development of programming skills is still an "infant field" (Pea, Soloway, & Spohrer, 1987), the literature is in accord with the Linn and Songer (1991) position. Studies describing students' misconceptions appear frequently. Claiming that students actively build a knowledge system of procedural skills and concepts, researchers have found that, rather than resulting from slips in mechanics of program construction, most faulty answers arise from systematic application of knowledge the student already has. The answers make sense when interpreted in terms of the student's current understanding (Pea et al., 1987).

Linn and Songer (1991) specifically identified the use of natural-world terms in scientific jargon as an inhibitor to the construction of science students' intuitive conceptions. In consonance, the programming literature also calls attention to the problems caused by the discrepancy between the technical and everyday meanings of many computer terms.

Principled understanding. In the third stage of the Linn and Songer (1991) model, learners acquire scientific principles. This acquisition, the authors hypothesized, is the result of schooling. It provides students with a reason to abandon their intuitive conceptions. Learners organize their intuitive conceptions into principles consistent with those held by experts. However, because there is often a large gap between intuition and scientific principles, scientific principles are frequently unattainable through unguided discovery.

In addition, Linn and Songer cautioned that students abandon construction and resort to memorization if they are presented with inaccessible principles. It is possible to speculate that learning to program is cognitively demanding precisely because the nature of the activity often precludes resorting to memorization.

It is perhaps the instruction characteristic of the model's third stage that makes it seem especially suitable for describing novice programming conceptions of the parameter construct. Self-taught (usually BASIC) programmers invariably write programs composed of unstructured spaghetti code. The structure of the Pascal language (as it is taught in introductory courses) effectively prohibits students from producing spaghetti code, but it does not force students to write modular programs. Self-taught Pascal programmers *sometimes* write one-module programs, but they *regularly* write programs that circumvent the parameter construct for communicating information between modules. Typically, these novices see no reason to abandon their intuitive conception in favor of the principle consistent with that of expert programmers, and they resist the change vehemently.

Summary. In short, Linn and Songer (1991) presented a model of conceptual change that is governed by a constructive process. According to this view, a number of factors affect the rate or pace of change. Schooling is of primary importance; students do not develop principled understanding of science or mathematics [or computer programming] through unguided discovery.

Related Research Literature

There seem to be two arguments for teaching programming represented in the related research literature. First, there are those who maintain that students who learn to program will develop problem-solving skills that will transfer to other situations. This argument prevails at the pre-college level. This strand of inquiry, although obviously important, is not pursued in this paper. Instead, this review focuses on the literature related to the position that computer programming should be taught for its own sake, with goals such as professional training or computer literacy.

Piagetian-related Studies

Formal thinking and success. The literature reports a number of Piagetian-based studies that seem to support the hypothesized connection between formal thinking and programming success (Cafolla, 1986-87; Fischer, 1986; Kurtz, 1980; Madison & Gau, 1993; Nachmias et al., 1986). In Fischer's studies, students' ability to construct complete combinations and permutations correlated strongly with the programming course grade. Hence, she concluded that students' ability to understand the relationship of parts to whole, and to systematically construct all possibilities, are critical for successful programming.

Kurtz (1980), who studied 23 FORTRAN students, administered a subjective, untimed test of formal reasoning that successfully predicted student performance in the programming course, correlating more highly with performance on examinations than with programming assignments. Furthermore, results on related formal reasoning test items displayed a weak correlation to students' prior mathematics, statistics, or logic background, suggesting that performance on the reasoning test was largely independent of specific school achievement.

The Cafolla (1987-88) study examined the relationship between the student's score on the final examination in a BASIC programming course and the student's verbal ability, mathematical ability, and intellectual development, as measured by Furth's (1970) Inventory of Piaget's Developmental Tasks (IPDT). The IPDT score and verbal ability were linearly predictive of success in the computer programming course. The study supported previous research claiming that not all college students are formal thinkers; of the 23 subjects in Cafolla's sample, 9 (39%) were found to be formal thinkers and 14 concrete thinkers.

The University of Wisconsin-Stevens Point (UW-SP) studies (Madison & Gau, 1993; Madison & Nazareth, in preparation) also incorporated the IPDT. The availability of construct validity (using traditional Piagetian interviews) and reliability information guided the instrument choice (Milakofsky & Patterson, 1979; Patterson & Milakofsky, 1980). The latest study, which included 307 UW-SP students, obtained results remarkably similar to those in Cafolla's study. Of the 172 students taking the introductory Pascal classes, 42% were formal thinkers. Of the 135 students taking advanced programming classes, 51% were formal thinkers. The UW-SP researchers also found statistically significant correlations between IPDT total scores and students' examination scores.

Other researchers, who studied pre-college programming students, found results congruent with the aforementioned studies. Nachmias et al. (1986) found their group of gifted fourth graders unable to grasp the concept of variables; sixth graders also had difficulty with the concept. Additionally, a study of high school students documented those students' difficulties (Kurland, Pea, Clement, & Mawby, 1986). After two years of programming instruction, high school students in the reported study had only a rudimentary knowledge of the programming languages taught. Few students could follow the flow of control correctly, and most could not systematically keep track of the values in variables. Observers noted that students used trial-and-error approaches and quickly asked for help. Programming success correlated strongly with students' ability to reason procedurally, decenter, translate word problems into symbolic equations, and analyze and design algorithms (Kurland et al., 1986): characteristics typically associated with formal thinking.

Concrete thinking and success. On the other hand, Kuhn, Ho, and Adams (1979) found in their study that 18 of 35 college students categorized as non-formal thinkers displayed either partial or fully formal reasoning when a problem was presented in a concrete rather than a written format. Madison and Gau (1993) also found only weak correlations between formal thinking and programming success in their study of college programmers. Additionally, Turkle and Papert (1990), reporting on their study of

Harvard University programming students, documented the programming success of students with a concrete style:

> When we looked closely at programmers in action we saw formal and abstract approaches, but we also saw highly successful programmers in their relationship with their material that are more reminiscent of a painter than a logician. They use concrete and personal approaches to knowledge that are far from the cultural stereotypes of formal mathematics. (p.128)

Multiple representations. Bruner (1960) claimed that any problem, idea, or body of knowledge could be presented in a form simple enough that any given learner could understand it. He advocated proceeding from the concrete to the pictorial to the symbolic representations, and considering the age and previous experience of the learners (Marchionini, 1985). Many students use visual imagery to reason; Battista and Clements (1991) urged us to respect this mode of thinking and to help students coordinate images with conceptual knowledge.

Researchers in the field of computer science education offer various suggestions for facilitating the absorption of complex concepts. Spohrer and Littman (1988) advised teaching students various representational devices such as graphs and rules to specify a problem. Basing their findings on clinical interviews, Perkins et al. (1988) described students' programming knowledge as fragile, due in part to students' lack of a mental model. The researchers employed a visual model of a "paper computer" to help students learn BASIC. Computer-generated visual animations of algorithms allow students to manipulate abstract representations (see Brown, 1988, 1989; Cicero & Groppe, 1993; Knudson, 1987, 1988; Multiscope, 1991; Naps, 1989, 1990a, 1990b; Popyack, 1988;

Sack et al., 1992). Others (Brown, Fell, Proulx, & Rasala, 1992; Taylor & Cunniff, 1988) use graphical feedback in college computer science courses in much the same manner as the Logo programming language's turtle graphics do.

Helping the learner to construct powerful representations is just one of the cognitive advantages Linn (1992) and Linn and Clancy (1990) claimed for their case study approach. The authors stressed the importance of multiple representations, "linking a verbal description, pseudocode, pictorial illustrations, dynamic illustrations, analogies, and Pascal code" (Linn & Clancy, 1990, p. 22). On the basis of their study, Turkle and Papert (1990) asserted that computer graphics, animations, and text can "provide a port of entry for people whose chief ways of relating to the world are through movement, intuition, and visual impression" (p. 131).

Schema-related Studies

Templates. Several schema-related studies performed between 1981 and 1992 were based on Piaget's notion of schemes and Anderson's (1983) idea of schemas. Soloway (1986) referred to schemes as domain-specific "chunks." The chunks form the programming plans or templates of code that expert programmers reportedly use to organize their knowledge of programs (Linn, 1992; Linn & Clancy, 1992; Van Merrienboer & Krammer, 1987). Programming plans, combined with goals and comments about the code, are schemas in concrete form. McKeithen, Reitman, Reuter, and Hirtle (1981) maintained that experts have better organized and more useful chunks of information. Their research revealed that experts surpassed novices when the experts recalled chunks of meaningful programs, but not scrambled ones. Similarly, Soloway and Ehrlich (1984) found that experts better recalled program code when it followed the "rules of programming discourse," that is, when it was plan-like.

Consistent with Anderson's schemas, many (Dalbey et al., 1986; Kurland et al., 1986; Linn, 1985; Sheil, 1981; Soloway, 1986; Soloway & Ehrlich, 1984; Spohrer & Soloway, 1986) suggested that experts have large libraries of stereotypical solutions (templates) as well as strategies for using them. "What," Soloway asked, "are the basic building blocks for analyzing problems and constructing programs? *Goals and plans*--stereotypical, canned solutions" (p. 851). Contending that learning programming language syntax and constructs does not cause novice programmers major problems, Soloway proposed that novices have real problems merging plans, or putting the pieces together (Soloway, 1986; Spohrer & Soloway, 1986).

Founded on programmers' hypothesized goal-plan cognitive structures, Spohrer and Soloway (1986) collected an entire library of errors in their efforts to develop a descriptive theory of "buggy novice programs." From this work and the work of others with Pascal programmers, Spohrer and Soloway ostensibly dispelled the "folk wisdom" that "*most novice bugs arise because novices do not fully understand the semantics of particular programming language constructs*" (p. 627), offering alternative "plausible accounts" for most novice errors. Since the authors reported that half of the errors either might be or certainly could be attributed to faulty understanding of language constructs, it is possible to contest their findings. More important for this study is the observation that nowhere in their discussion of buggy Pascal programming constructs did the authors mention parameter passing or procedures.

Programming language constructs. While the Soloway (1986) position that assembling plans (themselves composed of combinations of language constructs) causes more misunderstandings than the language constructs alone is quite reasonable, others (Linn, 1985; Putnam, Sleeman, Baxter, & Kuspa, 1986; Kong & Chung, 1985) reminded the programming community that knowledge of language constructs is a necessary link in the chain of learning to program. Putnam et al., for example, asserted, "A major component of learning to program is learning enough about the specific constructs . . . to gain an understanding of the 'conceptual' machine underlying a programming language" (p. 460). The authors reasoned that a programmer cannot design a program that uses IF (an implementation of a selection control structure) statements if the programmer does not understand how the IF statement works.

Putnam et al. (1986) drew their conclusions from a qualitative study of 96 high school BASIC programming students. The researchers interviewed 56 students who predicted program output and constructed simple programs. The investigators--who analyzed tape recordings, written notes, and responses generated during the interview--found eight categories of misconceptions of BASIC language constructs. The constructs listed included variables, repetition, and selection control statements. Once more, the authors did not mention the GOSUB statement (the BASIC language implementation of the Pascal procedure concept). In a similar study, the same group (Sleeman et al., 1986,

1988) described high school Pascal programmers' errors. Their appendix of constructs studied alludes briefly to procedures; it does not mention parameters.

Others researchers have also studied students' understanding of language constructs. Samurçay (1985) investigated beginners' looping (repetition) strategies. Du Boulay (1986) investigated problems with variables and flow of control (selection and repetition control structures). Kurland et al. (1986) studied selection and looping constructs. The Kurland group attested that programmers must have a good model of the structure of the language, pointing out that one cannot write a modular program without understanding how control is passed in the language, and how the logical tests that control the selection and repetition control structures operate.

Stressing the need for program readability and reader expectations, Joni and Soloway (1986) proffered a catalogue of programming discourse rules. All programs, they suggested, should follow these conventions or discourse rules. In their moderately extensive discussion of parameters (the only one found in the literature), the authors cautioned regarding the inappropriate use of VAR (variable) parameters. Joni and Soloway offer that inappropriately VARing parameters misleads program readers, makes the inappropriately VARed variables more susceptible to being "clobbered," and the program harder to debug (p. 115-117). In a footnote Joni and Soloway outline their suspicions:

> We suspect that many novice programmers have difficulty with VARing variables because they have misconceptions about argument passing in general. In addition, they often have an unclear picture of memory storage

> and retrieval, which contributes to their difficulties in VARing variables. (p. 115)

It seems reasonable to conclude that Joni and Soloway's observation is correct, that students' lack of an accurate mental image of the way in which parameters are passed in memory inhibits their understanding of the parameter construct.

Effective instruction. The Linn group (Linn, Sloane, & Clancy, 1987; Sloane & Linn, 1988) conducted a descriptive study of naturally occurring instruction in advanced placement (AP) Pascal classes. Armed with the goal of developing a theory of instruction, the researchers attempted to identify instructional strategies that promoted proficiency in designing solutions to programming problems. They found that three factors influenced learning: on-line access (time spent at the computer), explicit instruction in applying concepts and problem-solving strategies, and informational feedback on problem solving. The authors expressed little fear that too much explicit instruction would stifle the development of students' problem-solving skills, because students in an AP course typically spend much time working independently at the computer. However, the authors stressed the need for instructional balance, as small group and individualized instruction correlated more strongly with successful program creation than large group lectures.

Students learned better when they were explicitly taught than when they were left to develop independently (by discovery), a recurring pattern the Linn research group apparently found in the pre-college classrooms. In the authors' words, "Exemplary classes appeared to help students *construct* more productive understanding" (Linn et al., 1987, p.487).

Vygotsky-related Studies

Collaborative learning. Providing a basis for their research with collaborative computer programming environments, Webb and Lewis (1988) cited authors who theorize that group ideas may give rise to the conceptual conflict and resolution that is assumed to facilitate cognitive restructuring. Continuing, the authors offered that the resolution of disagreements that may occur during group interaction provides opportunities for members to "retrieve prior knowledge, evaluate their own and others' answers, ideas, and opinions, confront their own misunderstandings and lack of knowledge, and as a consequence, restructure their own thinking" (p. 181). Webb and Lewis went on to cite others who theorized that group activities provide opportunities for modeling and imitation, suggesting that successful peers may be better models than experts, especially for novices with doubts about their capabilities. The authors proposed that communication among peers may be more effective than teacher-student communication, that peers can be more direct, that they better "tune in" to each other's problem-solving processes and thinking, and thus provide more effective feedback.

Reports of successful collaborative learning, including some collaborative learning of computer programming, appear throughout the literature (Bredehoft, 1991; Cheny, 1977; Davies, 1988; Johnson & Johnson, 1987, 1990). However, it must also be stressed that group work does not guarantee learning for everyone (Lemerise, 1993; Webb & Lewis, 1988). Lemerise offered the general advice that group learning situations must be structured carefully if all participants are to benefit. Webb and Lewis concluded that positive effects do not take place if all students do not actively participate, if some lack the requisite skills, or if students fail to get help when they need it. After noting that the results of studies on group versus individual learning of computer programming are thus far inconclusive, Webb and Lewis suggested that it is more important to discover the factors that influence learning than to question whether individual learning is superior to group learning.

From the results of two Webb studies with adolescents (one using Logo and one using BASIC), Webb and Lewis (1988) alleged that "the most striking finding is that much of the planning and debugging carried out without the presence of an instructor was positively related to most achievement outcomes, whereas verbalizing the same kinds of behavior to an instructor was negatively related or not related to achievement" (p. 193). Not receiving help when requested was negatively correlated with success. The specific behaviors found to be correlated with achievement were giving and receiving explanations, giving and receiving input suggestions, receiving responses to questions, and verbalizing general planning and debugging strategies to peers.

As a closing note, Webb and Lewis (1988) cautioned that the Webb studies were correlational, not empirical, that they involved only one teacher who had a rather indirect teaching style, and that not all strategies may work with all students. Moreover, the studies did not investigate the mechanisms by which the peer interaction influenced learning. Additionally, it is pertinent to note that the intervention was very brief, and the subjects were volunteers. Furthermore, these studies, along with many others reported in the literature, were done with school children (adolescents and younger). The findings may simply not apply to adults enrolled in a professional program.

Application of cognitive apprenticeship. A decade ago, Deimel et al. (1983) stressed the importance of students studying non-trivial computer programs. These authors expressed surprise about the continued write-only tradition of computer science, posing the following question:

> Programs are the primary artifacts generated by programmers. They are bodies of what is quite properly called a language. They constitute the basic literature of programming. Is it not surprising that we attempt to teach composition in unfamiliar languages without encouraging reading knowledge of the literature in those languages? (p. 104-105)

Drawing upon both the Vygotskian-based cognitive apprenticeship model and Anderson's ACT* theory, a related line of investigation has been expanded in the years hence.

Van Merrienboer and Krammer (1987), for example, studied three different approaches to teaching computer programming: the expert, the spiral, and the reading. Basing their analysis upon John Anderson's ACT* theory, the authors suggested six tactics for teaching computer programming: a) presenting a concrete computer model, b) using programming plans, c) incorporating program design diagrams, d) including annotated, worked-out examples (examples that include explanations), e) providing practice, and f) furnishing task variation. The authors championed the reading approach because it encompasses teaching strategies better matched to the six tactics than the strategies associated with the spiral and expert approaches.

Van Merrienboer and Paas (1990) added further arguments for the use of annotated, worked-out examples. Noting that expert programmers are believed to have thousands of task-specific procedures available in their memory, the authors suggested that such detailed procedural knowledge is likely to be tacit and not easily verbalized. Teachers, as a result, may have difficulty explaining such knowledge. To combat the problem, the authors stressed the importance of providing annotated, worked-out examples for the students to refer to during practice.

Van Merrienboer and Paas (1990) and Van Merrienboer and De Crook (1992) cautioned that the worked-out examples must provoke “mindful abstraction.” The authors suggested using partially completed, well-structured, understandable computer programs. As the novices must design and generate code, they practice basic program generation skills. Additionally, the students must understand the program in order to complete it; hence engaging in such an activity promotes conceptual understanding as well. A Van Merrienboer and De Crook (1992) study empirically demonstrated the superiority of the program completion method over a program generation method.

Drawing upon extensive computer programming (already cited) and science research experience (Eylon & Linn, 1988; Linn & Songer, 1991), researchers have developed a set of case studies for use in introductory programming classes (Clancy & Linn, 1992a, 1992b; Linn, 1992; Linn & Clancy, 1990, 1992). A case study consists of a problem statement, one or more solutions, and commentary describing the design solutions. The authors suggested that the case studies provide opportunities for novices to "think along" with experts in posing and analyzing solutions, thereby engaging the students in apprenticeships with expert programmers. Using the case studies, students solve problems they could not solve alone.

Linn and Clancy (1990) are convinced their case studies will help students develop program design skills by: a) helping students construct a powerful representation of pertinent knowledge, b) reducing the task complexity for the learner, c) focusing on the problem-solving process rather than the solution, and d) nurturing the development of students' reflection and self-monitoring skills (p. 22). The authors also echoed the obvious, but sometimes unheeded, truism that program design skills must be explicitly taught; novices, they contended, do not absorb the skills by some osmosis-like process (Linn & Clancy, 1992a).

Basing their conclusions on the results of interviews with competent programmers, Linn and Clancy (1992) also concluded that experts do not use the linear top-down design process most classroom instruction preaches. In reality, the authors said, experts "engage in top-down, bottom-up, and middle-out design as well as lots of backtracking and revision of their original ideas," (p. 125) contending that "instruction that suggests program design proceeds in an uncomplicated, top-down fashion confuses and frustrates students" (p.125). Apparently Linn and Clancy were referring to all students, not just to bricoleurs.

Summary

In summary, research verifies that teaching and learning computer programming are cognitively demanding activities. Learners with a more analytic cognitive style who demonstrate characteristics typically associated with Piaget's notion of formal thinking seem to be more successful in the structured programming environment than concrete learners. On the other hand, concrete learners have been shown to be successful. Moreover, providing students with concrete and graphical representations appears to help both groups of students understand the abstract programming concepts.

Results of research focusing on the collaborative learning of computer programming generally echoed the findings in other areas. Novices, when provided with opportunities to interact, to verbalize their understandings, to evaluate their own and other people's ideas, and to observe more able peers, succeed more often.

Expert programmers differ from novices in their possession and use of an extensive repertoire of programming plans or templates. Effective instruction has been shown to help students acquire those understandings and templates. Among the strategies that have been found to help students learn is explicit instruction, combined with a balance between small group and individual instruction and large group lectures. Explicit instruction includes providing students with annotated, worked-out, partially-completed examples and concrete models.

The body of literature associated with teaching and learning computer programming documents the virtual absence of the topic of parameter (argument) passing from the research base. As a result, direction for the study reported in this paper was gleaned from the broader body of literature devoted to the teaching and learning of computer programming in general. Special attention was given to the various ways that learners process information and to the instructional methods that appear most effective in helping novices acquire the complex cognitive skill of computer programming.

CHAPTER III

METHODOLOGY

Introduction

This is a theory-driven qualitative study; it is an exploratory, descriptive study that draws from several methodologies traditionally employed in qualitative educational research. Ethnographic methods investigate participants within a relevant context to gain an understanding of some phenomenon from their point of view. Data collected from a variety of sources (observation, interviews, and documents) are then examined for constructs or themes (Burton & Magliaro, 1987-88). The data for the study reported in this paper were collected from all three sources.

Research on computer programming has often employed protocol analysis. During protocol analysis, data are typically gathered by recording participants' verbalizations as they perform a target task. Ericsson and Simon (1980) asserted that verbal data that simply relate the problem solver's thoughts are valid data sources and do not unduly interfere with the problem-solving process.

Sheil (1981) suggested that introspection while formulating a program segment is perhaps the most compelling subjective evidence of a programmer's skill. Putnam et al. (1986), in a descriptive study of BASIC programmers, asked students to trace programs and explain how the programs worked. The Putnam group used tape recordings, written notes, and responses generated during interviews to search for patterns of errors and misconceptions about the constructs under study. Perkins, Hancock, Hobbs, Martin, and

Simmons (1986); Sleeman et al. (1986); and Swan (1993) described a structured clinical observation method in which the experimenter presents the programming problem, observes the student's attempts to solve the problem, and probes for explanations of the student's intentions and ideas. The study reported in this paper followed the lead of the earlier computer programming research and incorporated protocol analysis along with the previously discussed ethnographic data collection methods.

General Design

The study employed a multi-case design, the unit of analysis being a novice programmer's understanding of the programming language construct of parameter passing. The unit is drawn from a general class of concerns with a novice programmer's conceptions of computer programming language constructs. A case study design was chosen because a qualitative design is ideal for understanding educational phenomena (Merriam, 1988). The author maintains, moreover, that the design is particularly suitable "for dealing with critical problems of practice and extending the knowledge base of various aspects of education" (p. xiii).

Importance of the Study

It is important to study this construct because, as noted earlier, the literature documented well the difficulties novices encounter when learning to program. Linn et al. (1987) proposed a chain of cognitive events leading to the development of problem-solving skills. Understanding the programming language's fundamental constructs was the first link in the chain. The literature also revealed numerous studies related to program design and several related to fundamental language constructs such as variables and conditional flow-of-control (repetition and selection) structures. What appears to be missing is a study of the construct of parameter passing.

Modularity is the first principle of structured programming (McIntyre, 1991). In structured programs, individual program modules are linked together to form a complete unit. Parameters provide the connections--the links--between the modules (Nance & Naps, 1989). Hence, parameter passing, the mechanism by which the various modules in a program share information, is a pivotal topic in an introductory course. Its mastery is crucial for a student's continued success in the discipline since virtually every program the student encounters henceforth will incorporate the construct. As Dr. William Wresch, chairperson of the Department of Mathematics and Computing at UW-SP elaborated recently:

> A complex program which consists of a single piece is incredibly difficult to understand. Programs are always partitioned into modules to make them more manageable. Novices have to learn this technique during their first semester or change majors. (personal communication, September 13, 1994)

Beaulieu (1990), arguing for the use of structure charts (a structure chart is to a program what an organization chart is to an organization) in introductory Pascal classes, observed that many students have difficulty understanding the concepts of value and variable parameters. Experienced programming instructors have confirmed the difficulty novice Pascal programmer experience when they encounter the parameter construct. The fact that high school Pascal courses frequently omit the topic attests further to the challenges it presents to novices.

Research Questions

The study addressed the following questions:

1. What action knowledge does the novice programmer bring to the study of parameter passing?
2. What intuitive conceptions does the novice programmer possess regarding parameter passing?
3. How does the novice express principled understanding of the parameter passing construct?
4. What is the relationship between a novice's conceptual and procedural knowledge of parameter passing?
5. What impact does a student's understanding of parameter passing have on the eventual success in an introductory programming course?
6. What impact does a student's understanding of parameter passing have on his or her disposition towards computer programming?

Influence of Theory

Several theories influenced the design a priori. First, Piagetian theory guided the purposeful selection of participants and guided the initial formation of two broad categories: formal and non-formal thinkers. Linn and Songer's model provided another dimension in which a participant's conceptions of parameter passing and related constructs could be categorized as action knowledge, intuitive conceptions, or principled understanding.

The work of Turkle and Papert (1990), in which the authors proposed that concrete and formal thinking are not stages of development but equally valid problem-solving styles, is crucial as it is from this perspective that the data were analyzed. Throughout the remainder of the paper, the learners whom Piaget described as concrete, and Turkle and Papert labeled bricoleurs, will be called contextual to avoid the pejorative connotation associated with the terms "concrete" and "bricolage." The learners that Piaget described as formal and that Turkle and Papert labeled planners will be referred to as analytical problem-solvers.

Setting

The study was situated at a state university located in a small city in the midwestern United States. The city would perhaps more aptly be described as a large small town where residents proudly boast of the clean streets and a friendly, hometown atmosphere. State University, which recently celebrated its centennial, is one of 13 four-year campuses in the midwestern state university system; it currently serves about 8,500 students. Most come from White middle or working class families, and many students return home on weekends.

Approximately 250 State University students currently declare CIS as their major or minor. As is traditional in a computing major, programming classes comprise most of the lower division course work. CIS 110 (Algorithm Development and Computer

Programming I--Pascal) is the first course in the CIS major, and several other disciplines require it as well. The three-credit course provides students with two 50-minute lectures and one 110-minute structured laboratory session during each week of a 15-week semester. The university offers four sections of CIS 110 every semester, each with an expected enrollment of 25 to 35. The data for the study were collected from students enrolled in one section of CIS 110 during a fall semester.

This class was chosen because CIS 110 is the Pascal class in which the construct of parameter passing is taught. The site was chosen because my position as a faculty member in the computing department at State University gained me initial access to the site. Additionally, as a regular teacher of the CIS 110 course, I had insider knowledge of the topics taught and typical student performance. Finally, the instructor, Dr. Greene (a pseudonym), and I had team-taught the class in the past. I was familiar with his presentation of the course content, and he was comfortable having me observe his class and talk with his students. Moreover, Greene's position as a tenured full professor provided him with professional security. Dr. Greene agreed that one of his sections would participate in the study prior to the start of the semester in which the data were collected.

Subjects and Sampling

Dr. Greene and I briefly explained the purpose and the requirements of the research to our respective students during the first class meeting of the semester. At that time, all students in all sections were offered the opportunity to switch to a research/non-research section. Students in the research section were also advised that they would be excluded from the study if they preferred not to switch to another section for scheduling or other reasons, but were disinclined to participate in the study. No students switched, and no students asked to be excluded.

During the second class meeting of the research section, Dr. Greene introduced me. I then explained the purpose and requirements of the study in more detail. As I passed out the informed consent forms, Dr. Greene signed the instructor informed consent form (see Appendix A), explaining that he was part of the study, too. Then he turned to me and, with a chuckle, asked: "What will you do if I refuse to sign?" "I'll be glad to do a reading at your funeral," I rejoined. Amid some laughter, all 30 students (10 females and 20 males) signed the participant informed consent forms (see Appendix B). The 30 students then completed a 21-question background survey (see Appendix C), and the last seven subtests of the Inventory of Piaget's Developmental Tasks (IPDT), placing their answers on Scantron sheets (see Appendix DS for selected items).

As elaborated in the paragraphs following, the results of the IPDT and the background survey guided the purposeful selection of the interview participants. According to Merriam (1988), purposeful sampling is based on the assumption that one desires to discover or understand a phenomenon. Thus, a researcher selects the sample from which he or she expects to learn the most. As a strategy for purposeful sampling, Merriam suggests choosing the typical case and the extremes. Accordingly, a sample of nine students (three with high IPDT scores, three with middle-range scores, and three with low scores) was chosen.

Inventory of Piaget's Developmental Tasks

In the several years preceding this study, more than 300 State University programming students completed the IPDT. Theory and classroom experience had already provided persuasive evidence that students with low scores struggle to acquire the same understandings of the abstract programming concepts that seem to come almost naturally for some students with high scores. Thus, it was reasonable to assume that students from the two groups might have differing conceptions of the parameter construct.

The IPDT, a 72-item multiple choice test, has been shown to be a valid (by comparing scores on the written group test with scores obtained from classic Piaget interviews) and reliable (using the test-retest and split-half measures) indicator of the Piagetian view of cognitive development (Milakofsky & Patterson, 1979; Patterson & Milakofsky, 1980). Because of a possible ceiling effect, the earlier studies followed Cafolla's (1987-88) example and used only the last six subtests (shadows, classes, distance, inclusion, inference, and probability), questions 49-72. This study did the same. (The students completed the angle subtest, questions 45-48, but the questions were not scored.)

Each subtest in the IPDT consists of four questions. According to the literature that accompanied the inventory, a subject must answer three out of the four questions for all six subtests to be considered a formal thinker. For purposes of this study, then, 24 was a perfect score. The IPDT scores in the research section ranged from 11 to 23. Upon the advice of Dr. Greene and the counsel of one doctoral committee member, a student with a score of 11 was eliminated as a potential participant. The student was a non-native English speaker, and there was concern that language difficulties would pose significant problems.

The most frequently missed question was #60, a somewhat ambiguous distance question. The question displayed several different rectangles (all with the same perimeter but with different dimensions) and asked: “Which rectangle is the largest or are they the same?” The correct answer, according to the scoring key, was the one with the largest area. However, 19 students selected “They are the same.” As no one asked for an interpretation of the term “largest,” it is reasonable to assume that some students believed largest referred to perimeter rather than area, in which case “the same” would be the correct answer. The next most-often-missed question was #55, a question on classes; of the 30 students who took the inventory, 16 missed it.

Background Survey

The background survey was originally conceived as a means of eliminating students with prior Pascal experience as participants, assuming they would have had prior exposure to the parameter concept. Question 10 asked, for example, about prior programming experience; then question 11 asked if the prior experience had been in Pascal. One student, who was being considered as a participant, answered ambiguously

("No" to question 10 and "Yes" to question 11). In order to clarify the situation, I called him. He had, it turns out, taken the one-semester, one-credit Pascal class at State University. The parameter construct is not included in the content of the course. Surmising that there might be other students who had correctly responded that they had completed a one-semester Pascal class, but who had not been exposed to the parameter concept, I called all students who were being considered participants on the basis of their IPDT score. Only those who verified that they had actually already studied the topic were eliminated as participants. No called students refused to participate.

Gender

The literature reported that gender does not correlate with programming success; it does, however, correlate with enrollment and retention, and State University echoes this pattern. The females who persist are at least as successful as the males; the low female enrollment and retention rates are troubling. Perhaps because gender does not correlate highly with programming success, the studies of student programmers' conceptual understanding do not generally report by findings by that variable. However, Linn and Songer (1991) reported that adolescent males and females construct quite different views of themselves as learners. Females, they claimed, do not agree with males about the nature of science and mathematics. The authors went on to say that females also report feeling unwelcome in domains constructed by males and perceived by males as being more male oriented. Turkle and Papert (1990) also found many women uncomfortable with the structured (Pascal) programming style. Indeed, if there are differences, those findings could inform practice. Consequently, each range of IPDT score included representatives from both genders in the interview sample. There were three females and six males, one female and two males from each IPDT range.

Participants

Before moving to data collection, this section introduces the participants. Table 1 summarizes the background of each as revealed by the IPDT and the background survey.

Table 1.

Participant Profiles

Name	Sex	Age	Class	Major	Minor	*	**	IPDT Score	Formal
Amy	F	< 21	Soph	Math	CIS	Y	Y	23	Y
Carol	F	21-25	Senior	Math Ed	none	N	N	20	N
Carl	M	< 21	Soph	Undecided	CIS	Y	Y	20	N
Jason	M	< 21	Soph	Paper Sci	none	N	N	22	Y
Moe	M	< 21	Fresh	Math		?	Y	23	Y
Rainbow	M	21-25	Soph	Undecided		?	N	19	N
Sammy	F	< 21	Junior	English	CIS	Y	Y	13	N
Sondra	M	< 21	Fresh	Paper Sci	none	?	N	17	N
Steve	M	< 21	Fresh	CIS	Business	?	Y	17	N

* Planned to enroll in subsequent programming class.

** Did enroll in the subsequent programming class.

All names are pseudonyms. During the first interview, the students selected their own pseudonyms, the only requirement being that the first letter of the pseudonym and the student's given name be the same. I considered changing some of the names to avoid possible confusion engendered by the choices. Carol and Carl are after all easily confused, Sammy is a female, and Sondra is a male. The participant choices were preserved because the pseudonyms may, in some small measure, reflect the various participants' personalities. A more complete "portrait" of the participants is included in the individual case study reports. Rainbow is not included in the study results. He ceased coming to class and submitting assignments some time between the first interview and start of the parameter instruction. He did not withdraw from the class and eventually received an F.

Data Collection Procedures

The first section provides a brief background in order to give a sense of the climate in which the data were collected. Descriptions of the various data collection procedures--observation, interviews, and document examination--employed in the study follow.

Background

During the fall semester in which the data were collected, I (the researcher) taught two sections of CIS 110 and Dr. Greene taught the other two. We followed an already established pattern of giving joint examinations and assignments, and using similar syllabi. As in the past, students were encouraged to seek help from whichever instructor was available. Greene taught the morning sections and held morning office hours; I taught the afternoon sections and held late afternoon (5:00-6:00 p.m.) office hours. Moreover, I typically worked in my office at least one evening a week.

State University's extensive network of computing facilities is a selling point of the school. It has numerous computing labs available for student use, including one in the Science building: a three-section lab that accommodates 75 students. The CIS students appear to prefer working in this lab over all others on campus. There are several reasons why this may be so. First, the CIS 110 structured laboratory sessions meet in the science lab; hence, students are familiar with the facilities. Second, all CIS classes meet in nearby classrooms, and CIS students tend to congregate in the general area. Third, the programming instructors' offices are located directly across the hall from the science lab, providing students working there with easy access to help. My office is directly across the hall from the Science building computer lab. Programming students have always sought my evening-hours help as I typically am the only programming instructor there at the time.

The semester in which the research data were collected was the first time that either Greene or I taught CIS 110 as a laboratory course. (It had until the past year been taught as a straight lecture course.) It was also the first time either of us used the textbook, *Turbo Pascal: Programming and Problem Solving* by Sanford Leestma and Larry Nyhoff. Opting not to use the accompanying laboratory manual, we developed our own laboratory activities. Thus, in many ways it was a new preparation for both of us, and most of the structured laboratory activities were being tried for the first time. Greene's laboratory sessions met at 8 o'clock and 10 o'clock on Friday morning; mine met at noon on Friday and at 3 o'clock on Monday afternoon. Throughout the semester, I regularly visited the lab on Friday morning to see if there were problems with the newly developed activities, problems that I might resolve before my classes met. When it appeared that Greene needed assistance, I stayed to help.

Moreover, the computing faculty at State University is a first-name faculty. During the class observations, Greene typically used the title "Professor" when he addressed me. Upon occasion, though, he called me "Sandy."

As a result of the circumstances--the teaching schedules, the office location, the frequent visits to Greene's class, and the informality--the students in research class knew me well and sought my help as regularly as did my own students. I did not refuse to help students just because they were participants in the study. (As helping was the established pattern of behavior, not doing so would have been an intervention.) Conversely, I did not seek out the participants to offer them additional help.

Classroom Observations

Overview. Classroom observations were included in the study primarily to provide a record of the instruction and the instructional climate. The videotapes, the student responses, and the field notes all provide evidence that Greene delivered the standard textbook content, and that the quality of the instruction was at all times acceptable. Pertinent aspects of the parameter-related instruction follow; specific participant reactions to the instruction are included in the individual case studies.

I observed all of Dr. Greene's classroom presentations where procedures and parameters were the topics of instruction (October 27 and November 1, 3, 8, and 15). All classes except one were videotaped. Typically I arrived in the classroom 10 minutes early to set up the video camera. On November 8, I was late; choosing not to disturb the class more than my presence was already doing, I relied on field notes. I observed and videotaped the structured laboratory on November 4. The class did not meet on November 10, and I did not observe the November 11 mid-term examination or the November 18 laboratory session.

Physical Environment. The short, wide classroom in which the lecture sessions were held accommodates approximately 35 people, with three rows each containing 11 or 12 desks. A blackboard spans the front of the room. One overhead projector, a screen, and a computer data display occupy center front, and a second overhead projector faces a screen mounted in the corner of the room.

As students typically do, the students tended to sit in the same place for each class even though they did not have assigned seats. In the front row, looking from the window to the classroom door, there were (in order) six females, two males, and one female. Amy and Sammy sat side-by-side in the middle of the front row. Jason and Sondra also sat in the front row, closer to the classroom door. The middle row held four males and then three females. Carol sat closest to the door, next to the video camera. In the back row, there were 11 males, among them Carl, Steve, and Moe.

The computer lab was arranged with five rows of six computers each. Seating arrangements varied from class to class, as students came in and sat at the available workstations.

Instruction. The planned syllabus followed the text closely. A brief background overview, basic Pascal, simple selection, and the WHILE . . . DO iteration structure preceded the procedure topic. Simple procedures were introduced first, followed by procedures with value parameters, and then variable parameters. Of particular interest is the fact that instruction on the procedure topic did not begin until midway through the semester and that FUNCTIONs (a second type of Pascal module) did not precede the procedure-parameter topic.

During the related classroom instruction, Greene used the blackboard and a lecture-large group discussion format exclusively. He began the instruction by motivating the topic of procedures thoroughly, sharing the reasons for modularization. He started with simple procedures, moved to value parameters, and then to variable parameters in a logical fashion. He used concrete analogies to great advantage. In one example, students assumed the different functions in an automobile factory; in another, he used access to a post office box as an analogy. The classroom presentations were lively, and the students seemed interested and involved.

Greene addressed the Pascal parameter syntax thoroughly. He addressed the terms formal versus actual parameters, explaining the distinction in terms of placement. He demonstrated the construction of the formal and actual parameter lists in several examples. Greene instructed the students to use different names for the formal and actual parameters, explaining that he was imposing the rule so that the students would avoid falling into the trap of thinking that corresponding actual and formal parameter names needed to be the same. His examples, moreover, inevitably used different names for the formal and actual parameters. In a very effective exchange, Greene asked students to think about how the association between actual and formal parameters could be made if they were not associated by name. It did not take long for a student to volunteer that the association must be made by relative position in the parameter lists. Greene said, "Think about it. Could it be done any other way?"

The textbook provided explicit rules for parameter list construction, and the partially completed examples in laboratory exercises (see Appendix E) provided practice using the rules. The original programming assignments (see Appendices F and G) and several examination questions (see Appendix H: questions 13, 14, 20, 21, and 29-31) also addressed parameter syntax.

As noted earlier, Pascal programs employ two kinds of parameters: value parameters and variable parameters. Leestma and Nyhoff (1993) motivated the need for parameters with the following:

> What is needed . . . are special kinds of variables in a procedure to which values can be passed from outside the procedure. In Pascal, these special variables are called **formal parameters** and are declared in the procedure's heading. Parameters that can store values passed to them but that cannot return values are called **value parameters** or **in parameters**. . . . Parameters that can both receive values and return values are called **variable** or **in-out parameters**. (p. 183)

Fundamental to incorporating the two kinds of parameters in a program is familiarity with the syntax. The instructor provided classroom instruction that reinforced the instruction provided in the text. He used the terminology provided there (*in* versus *in-out* parameters), and he covered the syntax of the differences carefully. Once more the assessment matched the instruction. Students could not construct parameter lists as required in the completion and construction tasks (see Appendices E, F, and G) if they did not know the syntax. Examination questions (see Appendix H: questions 4, 10-12, 15-17, 19, 23) also addressed the syntactical differences between value and variable parameters.

In contrast to the syntactical aspects of the parameter list construction, which conscientious students typically master without undue difficulty, novices often experience difficulties deciding when to use a variable parameter and when to use a value parameter. That decision must be based on the programmer's understanding of the semantic difference between value and variable parameters.

The Leestma and Nyhoff (1993) glossary defines a value parameter as a "formal parameter of a subprogram that is used to pass information to the subprogram but not to return information from it" and variable parameter as "a formal parameter of a subprogram that is used both to pass information to, and return information from, the subprogram" (p. 967). The authors imbed most of their text coverage on the semantic differences between value and variable parameters in an extended example that converts currency (see Appendix I for an excerpt). The text does include diagrams that depict the copy versus the shared memory-location perspective, and includes a verbal reference reflecting that view. In addition, Greene once more reinforced the textbook terminology. For example, he referred to variable formal parameters as being "associated with" actual parameters; moreover, he several times referred to a value parameter as a copy. But just as the text emphasized the one-way versus two-way communication view of parameters, so did the classroom instruction. Greene said several times, for example, "Value parameters are a one-way pass. With value parameters information only goes into the procedure; it does not come out" and "Variable parameters are a two-way pass; information goes into the procedure and comes back out."

During one class Greene choreographed a concrete simulation with the goal of helping students understand the differences between value and variable parameters. He called students to the front of the room to act as modules: a main module and several procedures. With the audience's help, the "modules" simulated the behavior of a modular payroll program. The modules received pads of paper that simulated variable parameters; the students could turn the page and write on the next page to send information back if they had a pad of paper. However, if the module received only a single sheet (a value parameter), there was no next page to write on. As the data will reveal, the concrete representation proved to be a crucially important instructional strategy. Once more, however, the simulation focused on the one-way versus two-way communication view of parameters.

The assessment reflected the instruction. In order to correctly complete the laboratory and construction exercises, students needed to understand that value parameters pass information into a procedure, but variable parameters return information from a procedure as well. Examination questions also addressed the behavior differences (see Appendix H: questions 2, 6, 7-12, 18, and 22).

The formal classroom instruction concluded on November 15 (the first class period following the test) when Greene reviewed the mid-term examination. During that class period, he spent considerable time tracing the code upon which examination questions 7-12 were based. As the participants were required to perform similar activities during some of the interview tasks, it is important to note that I did not observe any in-class code-tracing activity until Greene reviewed the examination questions and the correct answers.

In an introductory course, the parameter construct is sometimes--maybe more than sometimes--taught using a black box approach, one that emphasizes only the inputs (no VAR versus VAR) and the outputs (one-way communication versus two-way communication) and a process, without regard to how the process converts the inputs to the outputs. A State University advanced computing student who had taken her introductory courses at a sister institution recently described that situation succinctly with the following statement:

> I was told that if you put a VAR in front of the parameter, it comes back. If you do not put a VAR there, it does not. For now, take it on faith. You will learn more about how it happens in the advanced courses.

The following excerpt from *Structured Programming in Turbo* Pascal (1995), by L. Wayne Horn supports the student's statement:

> A value parameter is a parameter that the procedure uses as input; that is, value parameters are used to pass values to the procedure. We say that these values are **imported** into the procedure for processing. A variable parameter is used by the procedure to communicate with the calling program; that is, variable parameters are used to pass values from the procedure to the calling program. These are the **exports** of the procedure. Often, a variable parameter serves as both an import and an export. That is, a procedure may take the content of a variable parameter, process it, and return the modified value to the calling program. (p. 220)

In contrast, the Cooper and Clancy (1985) definition for variable parameter states:

> A *variable parameter* is an alternate name, meaningful only in the subprogram, for the variable that's supplied as its argument in the subprogram call. Changing the variable parameter *does* affect its argument, since they're exactly the same variable. (p. 72)

Given the extremes, the instruction provided in this study could be categorized as somewhere midway between the Cooper and Clancy (1985) and Horn (1993) coverage.

To summarize, much of the classroom and textbook instruction and the assessment used a black-box (communication oriented) approach to understanding parameter passing. It focused on the inputs, outputs, and the behavior of value parameters versus the behavior of variable parameters. What appears to have been under-emphasized was attention to the process that causes the difference in the value and variable parameter behavior. Under-emphasized, however, does not mean omitted. Both the text and Greene used phrases associated with the copy versus the shared-memory location view. Moreover, the diagrams embedded in the currency exchange program (see Appendix I) graphically illustrate the differences.

Participant Interviews

Participant interviews provided the main data source. Eight participants were interviewed three times. A fourth interview was planned in the event one of the participating students withdrew from the course during the interval between the completion of the instruction and the completion of the study; however, none of the participants withdrew. All interviews were audio-tape recorded and transcribed verbatim.

The first interviews tended to be fairly brief. The second two ranged in time from 30 to 90 minutes, depending on the participant. In order to preserve the participants' privacy, the formal interviews were arranged by telephone. However, the students occasionally stopped at my office door or in the hall to make arrangements. As Dr. Greene's office is nearby, it is likely he guessed the identity of some participants.

Interview One. Participant interview one (see Appendix J for the protocol), a semi-structured interview conducted in mid-October, focused on questions of prior computing experience. Samurçay (1989) stated that it is necessary to have a precise image of students' understandings about a concept in order to understand the difficulties encountered in teaching the concept. Thus, it is important to collect evidence about students' conceptions and the evolution of their conceptions during the learning process. The first interview thus also included brief questions related to parameters to establish a baseline understanding of the novice's conceptions prior to the parameter instruction. It also addressed the concept of a variable because understanding the variable construct is prerequisite to any understanding of the parameter construct, and is covered very early in the semester.

Interview Two. The second participant interview (see Appendix K for the protocol) took place just as the parameter instruction was concluding. A semi-structured interview, it commenced with questions about the participant's perceptions of the CIS 110 class up to that point. (In response to a query about the favorite and least favorite assignment, seven of the eight participants alluded to the out-of-class parameter assignment, the Burger Queen (BQ) assignment (see Appendix G).) The interview again included questions related to parameters. It also included an analogy that participants were asked to relate to the concept of parameter passing.

The second interview concluded with an analysis task in which students were given an almost-completed program and the output the program produced. Their task was to choose one missing procedure heading line from four provided (see Appendix L, Task 1) and explain their reasoning. According to Samurçay (1989), completion tasks differ from program construction because the completion tasks involve program understanding as a sub-task. Soloway and Ehrlick (1984) and Samurçay (1989) used a technique similar to that employed in this study. In the earlier studies, students were given an incomplete fragment of code with critical lines missing; the students were to fill in the missing lines.

Interview Three. Participant interview three was conducted several weeks later (See Appendix M for the protocol). By that time students had written several parameter-related programs and had taken a mid-term examination in which procedures was a major topic. The first interview-three task was completion-type activity as described previously (see Appendix N, Task 2). Using paper and pencil, students explained their reasoning as they constructed procedure heading lines for an otherwise completed program. The synthesis activity exemplifies the parameter-related decisions students must make when they write original programs.

Linn and Dalbey (1985) reported that teachers often assess knowledge of programming language features by "asking students to reformulate or change a language feature of the program so that the program does something different" (p. 192). In her studies, Webb merged two data sources: computer program listings and tape recorded verbal interactions (Burton & Magliaro, 1987-88). The second interview-three exercise (see Appendix O, Task 3) used the techniques reported in Burton and Magliaro and Linn and Dalbey. Seated at the computer, the participant converted a non-modular program into a modular program that incorporated the parameter construct. The program was saved on a computer disk each time the student compiled it (that is, each time the student believed the program was completed correctly). Intermediate versions of the program were preserved and the computer program listings later printed for use during analysis.

Linn (1985) stated that "students' knowledge of language features is often assessed with items asking them to predict how programs using the feature will perform" (p. 192). The last interview-three task (see Appendix P, Task 4) assessed students' understanding using that technique. The last task differed from the earlier tasks in that it was not similar to the tasks the students had seen during their instruction. It also differed in that it required the student to understand the shared memory-location perspective of the parameter process, while the previous interview tasks did not.

The third interview concluded by asking the students to imagine the procedure heading line of a well known, constantly used, but never-seen procedure, ReadLn, the procedure that transfers information from the keyboard into the application program. Never having seen the ReadLn's formal parameter list, the task required the participants to determine whether ReadLn's parameters are variable or value parameters based only on what they knew about the procedure's behavior.

Summary. The interview performance tasks did not measure all of the understandings or skills novice Pascal programmers are expected to develop during the first semester. Indeed, the selected tasks deliberately minimized the use of other programming constructs and skills, particularly those known to challenge novices. For example, the problems contained no loops, selections, or array processing. Moreover, only one interview exercise, the computer-assisted revision task, included program design; the design was as simple and straightforward as possible in a modular program.

Pilot testing of the interview protocols. The participant interview questions and several of the problem solving tasks were pilot tested during the semester prior to the initiation of the study. A State University computing faculty member, Professor Collier (a pseudonym), who was then auditing the CIS 110 course, acted as the testee. The Mona Lisa analogy (see Appendix J) was retained intact. Other tasks were modified as a result of the pilot study. For example, the pilot versions of the procedure heading line task (see Appendix L) and the paper-and-pencil construction task (see Appendix N) required only that Collier write down the reason for the choices. As the pilot test of those tasks yielded little information about Collier's reasoning, the participants in the final study explained their answers orally in a dialogue with the interviewer. As the next chapter reveals, the participant explanations proved to be valuable data sources.

The computer-assisted construction task was videotaped during the pilot study. The videotaping process, however, seemed intrusive, and the videotape was difficult to transcribe. In the final study, the task was audio taped quite successfully. The second analysis task (see Appendix P, Task 4) and the ReadLn exercise were not pilot tested.

The pilot study provided some guidance as to the conceptions potential participants might frticulate. It is important to note, however, that the CIS 110 class Collier was auditing at the time of the pilot study was not her first Pascal experience. She had taught the one-credit Pascal class for several semesters and had observed portions, including the parameter topic, of the CIS 110 class in the past.

During the interview, Collier commented that the textbook used in the course, *Turbo Pascal Programming and Problem Solving* (Leestma & Nyhoff, 1993), lacked an essential element she believed would greatly increase students' understanding: the use of concrete examples. The book, she noted, uses X and Y and N1 and N2 to designate variable names. She suggested, instead, the text employ "something that keeps, instead of just those letters." It does not matter how removed from the discipline the terms are, she asserted, offering "apples and oranges and peaches and pears, or black and white and red and blue," as possible alternatives.

Collier's words seemed to foretell the reaction of the students to the illustrations that proved most helpful. Her description of the instruction she provides on the variable concept (in the one-credit Pascal class) added further evidence to the value she places on concrete examples. One analogy involved the tale of a homeowner with three mailboxes.

"I have a mailbox that holds the Journal, a mailbox that holds The Shopper's Herald, and a mailbox that holds US mail," she explained. "Each mailbox knows exactly what kind of stuff it can hold. I tell them, 'The name stays the same on the mailbox, but what's inside keeps changing all the time,' " and added that the students enjoy and benefit from the analogy.

Collier also highlighted her use of "transparent envelopes" as an effective instructional technique for teaching students about variables. "The name stays the same on the outside," she summarized, "but using this transparent envelope idea lets the information come in and out." Although she confessed that many students initially reject the lesson as juvenile in nature, she added that a majority of them confide later that it is highly effective.

Turning to the parameter concept, Collier defined a variable parameter as follows:

> It simply means that we only get one box to share. We have only one spot for the two--the procedure and the main program--to share. Anything that's being done to the variable is also being done both locally and globally at that point--or in the program and in the procedure both--because we're both using the same spot exactly. We're sharing, and there's only one number allowed in there at a time, or only one value allowed in there at a time.

Collier's definition reflected the shared memory-location view of parameters, calling attention to the data storage notion of the variable that the procedure and the main module share. When questioned about differing names for the formal and actual parameters, Collier again replied with a comparison running along the same lines. "It doesn't make much difference," she responded. "It's just like at the post office. There are two sides in your mailbox, and it's the same thing here." She continued by recalling the practice of inserting mail in one side of the mailbox and retrieving it from the opposite side. "We're just using different sides," she reasoned, "but we're going to the same spot in the middle."

Collier's definition of value parameters was equally vivid. Her depiction of the process of passing information involved physically removing it and placing it "into another mailbox." Extending her metaphor, Collier referred to two locations ("mailboxes"):

> Say you put 4 in Apples. It's coming up from the main program, and we're into the procedure. I'll take the 4 and put it in Delicious. At that point, Apples just remains at 4, and it just fades out of sight; I never see it again. I just fool around within my little box that's got Delicious on it. That's it; anything that happens there just stays right there.

Collier's depiction firmly mirrored the copy idea of a value parameter; however, her comments ("Finally, hey? It's a wonderful feeling to finally see it!") reveal that she also struggled to attain this understanding.

Collier's elaboration on the Mona Lisa analogy also disclosed her skill in interpreting and relaying the sometimes perplexing reasoning inherent in parameter passing. Asked to summarize her understanding of the analogy, Collier first defined the value parameter passing: "the main program--that's the art teacher--has the Mona Lisa, the replica of the Mona Lisa." The teacher, she continued, hands the replica to the students, just as the value would be passed to the receiving procedure. Upon receiving the replica, the student "sets it up on the easel and then has his own easel beside and paints it as much as he can from that picture." Her portrayal continued with the student's copy going into his portfolio, and the teacher's untouched replica being returned to the supply closet. Her account concluded:

> Now the variable parameter passing would be that the person who owned the Mona Lisa would take it to the art shop. That person would actually touch the painting, actually change the painting, change the pigment, to restore it. And that would mean actually changing, so that the person would get back not the original Mona Lisa, but the Mona Lisa that's been taken care of, has had work done on it, actually physically had something done to it, something changed, to that Mona Lisa.

Collier's articulations firmly reflected the copy versus shared-memory location perspective of parameters. A review of the pilot interview transcripts and the problem-solving tasks, however, alerted me to the fact that none of the tasks required an understanding of that perspective. Task 4 (see Appendix P) was added to the final study as an attempt to probe for that understanding.

Instructor Interviews

Dr. Greene was interviewed twice (see appendices Q and R for the interview protocols), once shortly before the instruction on parameters began, and then again near the end of the semester after all other data collection was complete. During the latter interview, Greene shared his perceptions regarding the effectiveness of the various instructional strategies he had used, and the areas that posed the most difficulties for his students. The insights he provided helped guide the data analysis.

Speaking of the class's performance during the semester, Greene indicated that students were turning in all assignments, viewing the fact as a sign of success. Noting that during other semesters there "would typically be some people who, up until procedures, were making do." After that point, he offered, "they would end up with programs that might run, but wouldn't accomplish the task that was required," adding that he had not observed this phenomenon during the present semester.

Accounting for his students' high level of success, Greene credited the labs. In previous semesters, he recalled, students experiencing difficulty had only one source of assistance: the professor. "If 30 students needed help, they had to try to fit into one instructor's time," he pointed out. He added that the lab situation maximized learning potential, as, in his words, "we end up with students helping each other. So in one sense, there are 31 potential instructors."

Greene also noted the collaboration that he had observed among students working in the labs. Labeling the students "comfortable asking each other questions," he went on to mention situations where he witnessed higher-ability students who had finished their work moving to another pair to assist them, "acting almost like lab assistants."

Citing the completion-type laboratory exercises as a major reason for the students' high performance, Greene reflected on the strategy of providing students "with modules that work." The modules, he conjectured, increase students' success because the learners are able to focus on whether or not information is being transferred "back and forth" rather than wondering why the code did not work correctly.

Greene spoke of another strategy he felt had been successful:

> I still think my (and I've done it in the past and I've always liked it) notion of visualizing parameter passing in terms of sheets of paper. I use the strategy of a sheet for a value parameter and a pad for a variable parameter. The sheets would passed between people and then discarded; the pads would come back, that kind of thing. I think that has in the past been successful, and it was successful again this semester.

Sensitive to the fragile understanding some students still displayed at the semester's end, Greene observed the general tendency to overuse the VAR. Contending that the practice did not make a lot of difference "in the programs the students write during the introductory course because the value parameters are not modified," he remarked, "they tend not to be creating problems for themselves." Greene then astutely observed: "So they're not really forced to come to grips with that." Concluding, he said, "Obviously, if they don't pass the proper parameters and the procedure needs the information, the programs don't run."

Greene continued by expressing general satisfaction with the students' progress, stating that the majority of them understood the construct at an acceptable level. Although a small percentage of the students, he noted, still had but a rudimentary understanding of parameter passing ("they grasp parameters as a means of conveying information to the procedure, but they have not yet made a distinction between which have to be also conveyed backwards"), Greene pointed out:

> I have a few very good students who, at this point, seem to be able to write a procedure and rather quickly understand, almost without thinking about it, what parameters need to be passed, and more importantly to me, which ones need to be variable and which ones need to be value.

Greene's observations generally reflect the literature regarding the value of the structured laboratory environment, the collaboration, the focused completion exercises, and the concrete representations. Moreover, as the participant data will reveal, his views are in harmony with the thoughts expressed by the study participants, both in regards to the instruction that was most valuable and to the difficulties they were experiencing.

Documents

Academic transcripts, the background survey, the IPDT results, the written interview tasks, the printed program listings from the program revision activity, the mid-term examinations, the participants' written class work, and Dr. Greene's grade book were all potential document sources. The participants' written work was especially important as it often confirmed the participants' interview statements. Moreover, the programming patterns demonstrated during the interview tasks were frequently mirrored in the written work. In several instances, the combination of data sources--the participant's verbalizations, the solutions to a variety of diverse parameter-related tasks, and the written work--provided persuasive evidence for interpretations that would have been insupportable with fewer data sources.

Data Preparation

A hand-written data sheet containing pertinent information from the IPDT and the background survey was created for each participant. It contained such information as the participant's IPDT score, categorization as formal versus non-formal thinker, IPDT questions missed, major, minor, gender, age category, previous computing experience, previous Pascal experience, and expectations regarding success. The information on the data sheet was used in the initial search for categories.

The interview transcriptions, expanded observation field notes, and a methodological log were all prepared using the word-processing program, *Word for Windows* 6.0. Hired transcribers transcribed the interviews verbatim. To insure the accuracy of the record, I edited each transcription in its entirety. Every line of each interview transcription was coded with a participant identification code and a unique line number. The interview transcriptions and the *Turbo Pascal 7.0* program listings were preserved on paper and in machine-readable format. Handwritten notes were prepared from pertinent sections of the videotapes. In one instance, where Dr. Greene was explaining the code-tracing question from the mid-term examination, a 10-minute portion of the videotape was transcribed completely.

During the data collection period, Dr. Greene made available the work produced by all students in the research section. He generally kept the work in a cardboard box for transporting between the university and his home. I personally removed, photocopied, and then returned the participants' original assignments and tests to the box. The copies were hand coded as necessary.

As this is a multi-case study, the participant data were arranged according to case, using file folders. In each folder, the data were arranged chronologically. Maintaining the bulk of the data in machine-readable format allowed for electronic "cutting-and-pasting," while the coding scheme preserved identification of the original data source.

Data Analysis

Merriam (1988, p. 60) described a method of constant comparative analysis for use in qualitative studies. During the first stage of the process, incidents are compared and tentative categories developed or properties generated. Next, incidents are compared to the properties of the categories that were developed during the first stage. Third, the number of categories is reduced, hypotheses are proposed, and the data are reviewed to determine how closely they match the categories. Merriam (1988) stated that the "goal of analysis is to offer reasonable conclusions and generalizations based on the preponderance of the data" (p. 130): to discover categories or themes that emerge from "recurring regularities in the data" (p. 132).

Preliminary categories

As noted earlier, Piagetian theory guided the purposeful selection of the participants. During the informal analysis that accompanied the data collection, the theory also suggested early categories: formal versus informal thinker, and high versus medium versus low IPDT score. The selection of participants from both genders allowed for another category: male versus female. Previous programming versus no previous programming experience, and previous Pascal versus no previous Pascal experience suggested still other schemes. College grade-point average, major, and disposition toward mathematics were also considered. As the data collection continued, the preliminary categories all collapsed, as none of the categories seemed to show a relationship to the levels of understanding displayed by the participants.

All-inclusive category

A perusal of the interview protocols reveals no reference to the ReadLn exercise. The decision to include the exercise in the interviews resulted from a growing realization that an unmistakable pattern was emerging: a pattern which suggested that the participants were unanimously and consistently articulating a conception that reflected the one-way versus two-way communication view of the parameter process.

As the study progressed, I questioned whether the opaque view provided the students with sufficient understanding to visualize an unseen parameter list. If they were successful with the ReadLn exercise, I hypothesized, it would be reasonable to conclude that the students' reliance on the opaque view did not unduly hamper their understanding of the parameter construct. Conversely, if the students were not able to correctly identify the ReadLn parameters as variable, the fact would add support to my emerging hypothesis that a transparent conceptual model, one with fidelity to the process as well as the products, might improve novices' understanding of the parameter construct.

The startling failure of *all* participants to correctly solve the task (see Appendix P, Task 4) whose solution required the participant to apply the shared memory-location view of the variable parameter process eclipsed other preliminary findings and directed the initial stages of the formal analysis.

Linn and Songer categories

The Linn and Songer (1991) model of conceptual change provided an integrative categorization scheme. The first interview, conducted prior to the instruction, allowed for classifying the knowledge the learners brought to the parameter learning situation as action knowledge. Moreover, formal analysis revealed patterns of fragile understandings and misconceptions that could be grouped under intuitive conceptions. Principled understanding could serve as an over-arching category allowing for the examination of the students' conceptions from different perspectives. Learners, for example, could demonstrate principled understanding of the parameter process from either the copy versus shared memory-location perspective, or from the one-way versus two-way communication view.

Formal analysis revealed a second important dimension for examining students' principled understanding of the parameter construct. The participants employed problem-solving styles comparable to those reported in Turkle and Papert (1990); some displayed an analytical style, some a contextual style, and one seemed to move between the two styles with ease. The over-arching category allowed for attention to the different learning and problem-solving styles students employed.

Formal data analysis commenced in February after all interviews were transcribed. After reflection during the analysis, I constructed an "understanding continuum" along which the participants were arranged. Moving along the continuum from most principled to least principled, the participants, in order, were: Carol, Carl, Amy, Sammy, Sondra, Steve, Moe, and Jason. Although the placements remained stable, formal analysis continued until mid-June when I could provide convincing explanations for the appearance of the continuum.

Trustworthiness

Merriam (1988), citing Lincoln and Guba (1985), stated that one measure of a study's trustworthiness is its truth value or internal validity: how closely it reports what is really there. A researcher insures truth value by using techniques such as multiple sources of data, member checks, long-term observation, and peer debriefing. Trustworthiness for this study was established by using the technique of triangulation (collecting data through multiple data sources), by incorporating multiple cases, by spending extended time in the field, by employing member checking, by including peer debriefing, and by providing thick description. All participants were interviewed three times, during which they solved a variety of parameter-related problems. All classes in which the topic of parameter passing was the focus of instruction were observed. The instructor was interviewed twice, once just before the instruction and again near the end of the semester. The parameter sections of numerous textbooks and the parameter-related class work of all participants were examined. Member checking in the form of questions such as "This is what I heard. Is this what you mean?" and "Did you mean to say that . . .?" was employed. The department chairperson at the university in question acted as a peer debriefer throughout the data collection and data analysis processes, and Dr. Greene himself acted as a peer debriefer during the concluding interview. Finally, the findings were shared with the pre-service teachers enrolled in the computer science teaching methods class.

Merriam (1988) described the extent to which the findings can be applied to other settings (or in more traditional terms, replicated) with terms such as reliability, consistency, transferability, and external validity. The thick description found in this and the next chapter provides readers with a detailed view of the environment in which the study was conducted, enabling readers to reasonably estimate the extent to which the findings may transfer to other settings. Additionally, the purposeful sampling of participants encourages transferability.. An audit trail, which includes a methodological log and the safekeeping of all original data sources (videotapes, audio tapes, field notes, expanded field notes, documents, transcriptions, and computer files) ensures dependability and confirmability.

Summary

This study employed a theory-driven, qualitative design to investigate novice Pascal programmers' understanding of the parameter passing construct. The bulk of the data was collected from eight college students enrolled in a state university introductory computer programming course. Observations of the programming instruction, interviews with the course's instructor, and examination of pertinent documents provided additional data. Trustworthiness was established using a variety of techniques including triangulation and thick description.

CHAPTER IV

RESULTS

Introduction to the Case Study Reports

The individual eight case studies are arranged along a continuum developed after considerable reflection during formal data analysis. The continuum begins with the two students whose understanding appeared principled (Carol and Carl) and moves to the three students who appeared to harbor fundamental misconceptions of the parameter construct (Steve, Moe, and Jason). Between the two extremities lay the three students whose understanding showed varying degrees of fragility (Amy, Sammy, and Sondra).

Commencing with a capsule of the individual case, each narrative then paints a verbal portrait of the participant, with some attention to his or her background. The report next considers the participant's reactions to the CIS110 course, attempting to provide information concerning, for example, the topic the participant considered most challenging (inevitably parameters), favorite and least favorite assignments, and pertinent data on the class work. Following a segment specifically devoted to participant remarks concerning parameters, the account considers--in order--the participant's response to the Mona Lisa analogy (see Appendix J); problem-solving efforts on Task 1, Task 2, Task 3, and Task 4 (see Appendices L, N, O, and P); and the participant's interpretation of the ReadLn parameter list.

Case Study: Carol

A Capsule of Carol

Carol demonstrated a principled understanding of the parameter construct from the communication perspective. Although she displayed a faint inclination toward the shared memory-location outlook when she used the words "copy" and "original" to describe value parameters, the bulk of evidence shows that Carol relied on the communication view. She articulated the terminology of that perspective effortlessly and succinctly, she correctly analyzed programs that did not require the shared memory-location view, and she constructed modular programs without errors.

According to the IPDT, Carol was categorized as a non-formal thinker, but the problem-solving style she demonstrated and verbalized was consistently that of an analytical learner. The mental models literature (Mayer, 1982, 1985, 1989) and Turkle and Papert (1990) suggest that people with an analytical style are comfortable with a black box model; they are quite happy to use objects without fully understanding all the details. As Carol appeared to have been unhampered by her reliance on the rule-driven, abstract parameter passing model, the case of Carol would seem to lend support for the position.

Background

A female in her early twenties, Carol is a senior-standing transfer student who is majoring in mathematics education. As might be expected from her major, Carol indicated that she enjoys mathematics and does well in it. Her transcript revealed her to be a B/C student overall and in the major. A computer user since middle school, Carol had copied BASIC exercises that came with an early computer, had taken a high school computer applications course, considered herself comfortable with word processing, and had previously taken State University's campus computing facilities course. At State University, mathematics education majors are required to take three credits of computing, but the three credits need not be CIS 110. Carol's reason for enrolling was, in her words, "to learn something new," and she expressed her desire to succeed in the course. Despite the slight initial concerns, she earned a high A.

Carol scored 20 on the IPDT, missing questions 49 (a shadows question), 54 and 55 (classes questions), and 60 (a distance question). The two mistakes on the classes subtest prevented her from being categorized a formal thinker.

Reactions to the Course

At the semester's outset, Carol indicated that she did not plan to continue in computing, offering the following explanation:

> Honestly, I had a friend who was a CIS major. She stumbled through 110 and ended up changing majors. She was my roommate at the time, and I would see her with all the problems.

Describing her perception of the first part of the course, Carol said, "So far, it's been going real good. I thought it would be impossible, but it's real easy." Midway through the semester, Carol said, "I really enjoy the course. It seems like it's not too hard for me.

In fact, it's the class I'm doing the best in." The following semester, Carol commented on her decision not to continue in CIS:

> Halfway through the semester I was kicking myself for not taking it [CIS 110] earlier. I do not have a minor, and I definitely would have chosen CIS. I enjoyed the course more than any other I have taken since I've been here. I student teach next semester, and then I'm done. But I'm thinking that once I've started teaching, I'll come back and pursue the minor.

Carol's observations have implications for the question of a relationship between programming success and a positive disposition toward the subject.

When asked to describe her favorite element of the course, Carol was initially undecided, then claimed she liked "all of it" except for completing the laboratory exercises in the lab. "I prefer to work at home. At the beginning it helped when I really had no idea about it. Now I'm better off doing it at home on my own time." The only participant who commented negatively about the structured laboratory component of the course, Carol's observation that she benefited from the experience when she was unsure of herself is in accord with the less secure students who strongly endorsed the labs.

Speaking of the assignments, Carol offered that she enjoyed figuring out how to solve the problems. She said, "I feel like I really accomplished something when I start out with a problem, and I've got a program that produces the answer to it. It feels like a real accomplishment." The BQ assignment, which she described as challenging, was her favorite. "When my original idea of how I was going to write it didn't work out, I was able to figure out where my problems were," she claimed, suggesting that she enjoys controlling the machine, as did the planners in the Turkle and Papert (1990) studies.

Carol, who consistently demonstrated an analytical problem-solving style, also volunteered that she liked the graphical banner assignment. Her words underscore the supposition that contextual learners are not the only ones who benefit from concrete representations. A review of Carol's parameter-related class work reveals that it matched her assessment of her performance. It was consistently done correctly.

Parameters

When asked which concept she had thus far found the most challenging, Carol replied, "The difference between variable and value parameters. As I was doing the banner and the revised banner, all of a sudden it just seemed to click. I knew what it was." Carol's forthcoming description of value parameters illustrates her accurate understanding of the concept. "The information in the variable is passed into the procedure, but is not passed back out," she said, and described changes to value parameters as follows:

> It's just left in the procedure because the main program has the original value. The procedure gets a copy of the original, and that's changed, but it's not passed back to the main program. So the original value stays in the main program.

Her descriptions demonstrated a firm understanding of the value parameter concept, reflecting both the shared memory and the communication perspectives. When asked for a definition of variable parameters, Carol replied, "Information that is passed into the procedure through the variables. It can be changed, and then it is passed back out to the main line."

Carol offered that the in-class demonstration provided her with a mental model of the way information is passed to a procedure, saying, "I just kind of remember back to that in-class exercise that he had with passing the pads of paper." Her image of how information is passed into the procedure and back out, however, appears to lack that clarity. Her reply to the question "Do you have an image of how information is passed into the procedure and back out?" was "Not really. It's not really pictured or anything except for the in-class simulation." Carol's reference to the in-class simulation of the parameter process affirms the value of concrete representations.

Analogy

In the analogy segment of the interview (see Appendix K), Carol proved her ability to conceptualize and describe the action of a variable parameter situation. When called upon to make a comparison concerning situation two, she correctly responded that it reminded her of a variable parameter:

> How it's changed and then passed back to the owner, like the main program. Situation one is more like the value parameter, where the painting that the student made stays there. It's not passed back.

Carol's response aptly described the action of a variable parameter situation. The procedure changes the painting and passes it back to the calling module in its changed condition. Carol's interpretation identified the student-created paintings with value parameters; thus her analogy was correct in the sense that the newly created novice paintings (changes made during the procedure) were not passed back to the instructor. Carol's analogy reflected the communication view of the parameter process; however, it did not incorporate the notion of "copy" that she used earlier when describing a value parameter.

Task 1

When presented with the analysis task, Carol first looked to the main program and correctly observed, "The I in the main program is 16, and the output for both numbers is 16. So when you swap them, the I remains 16. The J would be changed to 16." Carol then looked to the details of the procedure, traced the code without hesitation, and chose the correct procedure heading line. She also recognized that she had just seen something similar on the test and in a previous lab exercise.

Task 2

Interview three, conducted on November 24, began with the paper-and-pencil construction task. After studying the program briefly, Carol confidently constructed correct procedure heading lines, giving appropriate reasons for her choices. She offered that the formal parameter in the GetInput procedure needed a variable parameter "because the dimension has to go back to the mainline program." She identified the CalculateVolume parameters as value parameters "because the information comes from the main program and is used to calculate the volume." "Volume," she said, "needed to be a variable parameter because it is given back to the main program."

Evidence of Carol's analytical problem-solving approach surfaced during the interview task. She addressed the problem in a structured, top-down fashion that reflected the classroom instruction. In a structured approach to programming, students are encouraged to decide first what a program's input and output should be, and only later develop a process that converts the input to output. Specifically Carol, as she analyzed the problem, looked first at the output procedure even though it physically appeared second on the page. She moved next to the input procedure that physically appeared third on the page. She examined the calculate procedure last, even though it physically appeared first.

Task 3

Carol methodically converted the one-module program to a modular version without difficulty. She routinely used different names for the formal and actual parameters, claiming it was easier for her to understand if the names were different. Carol addressed the procedures in the logical order of their execution, and carefully checked the parameter list order and names before attempting to compile the program. Her first attempt was successful; moreover, she asked for paper and pencil to verify that the program results were correct.

Task 4

When given the code tracing task, Carol looked first to the main module, saying:

> N equals 3. When you call the procedure, you put in N twice. In the procedure, the first N correlates with A, and the second N correlates with B. They're both variable parameters. In the procedure you add 1 to A, and you add 1 to B. You end up getting A is 4 and B is 4. Because

> they're variable parameters that are passed back, the main has that value for N. With WriteLn (N), you get 4 as your output.

Carol's answer reasonably reflected the communication viewpoint of the parameter process, the perspective she consistently expressed. Although Carol had earlier used the terms "copy" and "original" in reference to value parameters, she had never articulated the shared memory-location outlook in reference to variable parameters. She solved the problem appropriately, in terms of her understanding.

ReadLn

Without hesitation, Carol offered that a ReadLn formal parameter needed to be a variable parameter because "you use the information later on in the main line." As Carol was not asked to apply her understanding in an example, or asked how the outcome of a procedure call to ReadLn would differ if the parameters were value parameters, it is not known whether she could correctly apply her knowledge.

Case Study: Carl

A Capsule of Carl

Carl demonstrated a principled understanding of the parameter construct from the communication perspective. He facilely expressed the terminology and constructed programs modular programs without significant errors. The data disclose that Carl relied on the communication view for value parameters; he never used the term "copy." Most of Carl's reports suggest that he also firmly relied on the two-way communication view of variable parameters, although his closing comment during the second analysis task hinted at some perception of the shared memory view. If the interpretation is correct, then Carl seemed to be integrating the two views.

Carl's IPDT score of 20 categorized him as a non-formal thinker. However, his problem-solving approach on the interview tasks displayed an analytical style; without variation, he looked at the broad view first and attended to details as necessary. Although Carl did say he imagined variable parameters as a chalkboard and did once allude to the in-class simulation, he claimed to have no concrete image of value parameters. Lack of such an image seems not to have impeded his understanding. After examining Carl's performance and his responses, it appears safe to assume that his case supports the theory propounded by Turkle and Papert (1990); he was unhampered by the rule-driven communication model.

Background

Carl is a male who is under 21. A sophomore without a declared major, Carl named CIS as his minor in the beginning-of-the-semester survey; he also indicated that he was uncertain whether he would take the subsequent computing course. He did enroll in and complete the next class. Carl has been programming for some time; his father, who has programmed professionally, began teaching him to program when he was in the fourth or fifth grade. The youth took a computer applications course, which he described as simple, when he was in high school. While he did study Pascal briefly while in high school, he indicated that the students had done no more than copy the instructor's programs, exercises no more than a half-page long. Describing himself as confident in his success in the programming course (he did earn an A), Carl also claimed that he enjoys mathematics and does well in it. Carl's college performance thus far would categorize him as a B/C student.

A non-formal thinker who earned a score of 20 on the IPDT, Carl missed questions 59 and 60 (distance questions), 61 (an inclusion question), and 70 (a probability question). The two missed distance questions prevented him from being classified a formal thinker.

Reactions to the Course

During Carl's second interview, on November 17, he described the course as interesting, "not too hard." Reflecting Turkle and Papert's (1990) description of their planners, he added that "learning how to make the computer do what you want it to" provided him with a sense of accomplishment. Carl said there had been no assignment that he would describe as either favorite or least favorite, commenting that they all seemed pretty relevant. A review of Carl's work suggests that he did initially have difficulties with parameters. His first procedure lab (Lab 7) contained numerous parameter errors, including several instances of value-variable mistakes. He also missed approximately 10 procedure-related examination questions on the November 11 test. However, Carl's BQ assignment, although it contained design and style flaws, passed parameters appropriately, and consistently used different formal and actual parameter names. Moreover, the second procedure laboratory exercise (Lab 9) was completely correct.

Parameters

Asked about procedures with parameters, Carl replied, "It took a little while to actually grasp the variable-value parameter type thing." He explained: "You're doubling your variable ratio because you have different variables in your procedures and in your main line, just keeping them straight in your mind." Carl's statement agrees with the Van Merrienboer and Krammer (1987) position that some novice programmer errors are due to information processing overload. The authors explicitly named top-down design techniques--to which the procedures and parameters are fundamental--as a troublesome area.

Carl continued to reflect on his parameter problems, naming specifically having to "decide if you wanted the number back out." He offered:

> At first that caused a problem for me, but then Dr. Greene did that little skit, the simulation. It seemed menial at first, but it actually made sense after a while because I found myself thinking notepad or paper. So it actually did help.

His statement testifies to the value of concrete representations. Echoing the terminology of the communication perspective, Carl offered that changes made to variable parameters hold, while changes made to value parameters do not. Asked what guided his thinking about when a variable parameter is appropriate, Carl aptly replied, "Is the variable going to be changed in the procedure, and do I need the change?"

Analogy

Presented with the Mona Lisa analogy, Carl responded without hesitation, connecting situation two with a variable parameter, stating, "They send it back, it gets changed, and it comes back changed." Situation one he connected with value parameters, offering, "The students paint it and change it, but when it comes out, it's still the original. The original is still there." Carl analogized appropriately, given the communication perspective. In situation two, he connected changes being made to an original painting with variable parameters. His response to situation one, where the original is intact--where actions of the procedure do not change the original--was consistent with his earlier statements.

Task 1

As Carl attempted to choose the correct procedure heading line, he looked first to the procedure lines. He immediately eliminated procedure line two (the one with both value parameters), saying, “One of them gets changed, and the other stays the same.” Asked what the procedure accomplished, he unhesitatingly replied, “It just rotates all the values. I suppose I should look at it first before I assume. Yes, it just rotates the values.” Carl then looked to the details; he correctly traced the values and chose procedure line one. Congruent with the characteristics Turkle and Papert (1990) attributed to analytical learners, Carl easily accepted the swap procedure without first examining the details.

It is perhaps relevant to note that the test and Greene’s extended review of code-tracing questions similar to the interview task preceded Carl’s second interview. Had the interview occurred earlier, Carl’s understanding might have been less secure.

Task 2

Interview three, conducted on November 22, began with the paper-and-pencil construction task. Carl first determined the procedure functions and named them accordingly. He turned first to the GetInput procedure and correctly decided it needed a variable parameter because “you need those to come back into the main line because you have to send them somewhere else.” He first chose an arbitrary name for the formal parameter. After looking at the ReadLn, he correctly changed the name to Dimension, commenting “I didn’t even see that before.” Carl next worked on the CalculateVolume procedure. He completed the task correctly except for the ordering of the parameter list in the CalculateVolume procedure. As he had done in previous task, Carl first gained a grasp of the overall task; only then did he look to the details, and apparently only then when he felt it was necessary. Although the interview transcript does not disclose precisely the order in which Carl examined the procedures, it is certain that he followed the logical order of execution rather than the physical order in which the procedures appeared on the page.

Task 3

Carl accomplished the modularizing exercise easily, the only errors being two misspellings. It seems logical, therefore, to assume that his occasional errors seemed to be mistakes of inattention rather than errors in understanding. One might speculate that Carl reduced his information-processing load by relying on the compiler to find syntax errors. It may be that his previous programming experience made him more aware of the kind of assistance the computer could provide.

Task 4

Given the code tracing exercise, Carl first made sense of the program, then attended to the details of tracing the code. He summarized his understanding:

> I can tell just by looking at it that it's 4, but I would set up a trace. A and B, the original value is 3 for both of them. It goes through the procedure once, adds 1 to both of them, and ends, and then WriteLn (N). I'm not sure what N is. N is both of them, really, but they're the same. So it would display 4. I'm a little confused by the one variable, though.

One of the few participants who alluded to the storage component of the term "variable," Carl had earlier defined a variable as "something that holds a value in memory." He had also said, "For variable parameters, I picture like a trace. It's like a little chalkboard. You just cross out what you had before." The task, in which "N is both of them," appeared to generate a small amount of cognitive dissonance, perhaps causing Carl to question his chalkboard image. That both parameters returned the same value (4) may have allowed Carl to reconcile the incongruity without abandoning his chalkboard image. If this is true, then a better task would be one in which A and B are not changed by the same value. Carl's reference to "N being both of them," and the statement of confusion when one variable was passed twice, suggest a glimmer of understanding concerning the parameter construct from the shared memory-location perspective.

ReadLn

Asked to identify the ReadLn parameters, Carl replied without hesitation, "A VAR parameter, because if it wasn't, there would never be anything in that variable." Carl's description implied that he understood the effect of value parameters; he was not asked to apply the knowledge.

Case Study: Amy

A Capsule of Amy

Amy was generally successful constructing modular programs. Her level of understanding was sufficient to permit the regular use of different formal and actual parameter names. As further evidence of her comprehension, she repeated the words of the communication perspective with understanding. Moreover, once she related the problem to her prior experience, she successfully analyzed the program in the first task. Additionally, she offered a reasonable solution to the second analysis task. During the hand-construction task, Amy correctly selected between value and variable parameters, giving appropriate reasons for her choices. However, she seemed to rely on the computer to assist her with the value versus variable parameter decisions during the computer-assisted task.

Amy struggled to envision the unseen ReadLn parameter list. As she admitted to having no mental image of the parameter process (other than the in-class simulation), it may be that her lack of a concrete mental model inhibited her from imagining the parameter list.

Amy is a formal thinker. It is possible to conjecture that her formal thinking ability allowed her to move between the top-down and bottom-up approaches, as she seemed able to do. Employing both analytical and contextual problem-solving styles upon occasion, Amy demonstrated a flexibility consistent with Linn and Clancy's (1990, 1992) observation of expert programmers. Alternatively, it may be that Amy exemplifies the Turkle and Papert (1990) bricoleurs who resorted to analytical problem-solving styles to succeed in a structured programming course. Amy's problem-solving approach on most of the interview tasks, her partiality to the concrete banner program, and the self report of her BQ assignment efforts support this interpretation. If this rendering is correct, it is reasonable to hypothesize that Amy would have profited from additional exposure to the transparent, concrete, shared memory-location view of the parameter process.

Background

A female under age 21, Amy is a sophomore who has declared a mathematics major and a CIS minor. As might be expected from the major, she reported that she enjoys mathematics and does well in it. Her academic transcript generally supports her assessment; it would place her as a B student. She took the one-credit Pascal class, which she described as easy, during her first semester at State University, and earned an A. Amy volunteered during the first interview that she had decided to minor in computing approximately half way through the earlier programming course. At the CIS 110 semester's outset, Amy said she hoped to do well in the class, but had some concerns about her eventual success. She did earn an A. Amy is a formal thinker, with an IPDT score of 23. She missed only question 55, a classes question.

Reactions to the Course

At the November 9 interview, Amy described the class thus far as challenging. Observing that she enjoyed trying to figure out answers and working for outcomes, she said of computer programming:

> I like to put a program into a computer and have it do things and print out something completely different from what it was given, to make it myself.

Amy's description of her interaction with the computer is remindful of the Turkle and Papert (1990) planners. It also indicates her image of success. She described the concrete banner as her favorite assignment thus far and the BQ assignment, which she described as "nerve racking," as her least favorite. Elaborating on her difficulties with it, she said:

> First I tried modifying a previous one. I thought that would be easier than just starting from scratch. I did all this stuff to it, and it didn't work. So I just threw that one out. I ended up doing it three times before I got to my final one.

Amy attributed her early difficulties with the assignment to parameters, and described her successful effort:

> The last time I did it, I wrote down each of the variables, and I wrote down next to it, everything that went with it. I think because I organized on paper, it was a lot easier.

Amy's description of her initial approach to the BQ assignment is reminiscent of the Turkle and Papert (1990) bricoleurs. She constructed her BQ program "by arranging and rearranging, negotiating and re-negotiating with a set of well-known materials" (p. 136), in this case the Flipper program she had written previously. As with some of the Turkle and Papert bricoleurs engaged in a structured programming course, Amy found herself more successful when she adopted a more analytic strategy.

Other evidence of Amy's sometimes contextual problem-solving style emerged as the interview progressed. For example, Amy noted that although her BQ assignment had not yet been returned, she believed it produced the correct answer. She admitted, though, that she had not verified the calculations. In her words, she trusted her work, believing that after all the time she had spent on it, it must be correct. In contrast to her earlier statements, this observation suggested Amy viewed her relationship with the computer almost as a partnership, and that she was trusting the machine to do its part.

Except that she used variable parameters in the output procedures, an error Amy said she had become aware of after the assignment was submitted, the parameter-related portion of her BQ program was correctly done. In every instance, she chose a formal parameter name that differed from the actual name. Moreover, Amy's laboratory exercises were well done, and she missed no parameter-related questions on the test two days later.

Parameters

Accrediting the ease with which she mastered the topics before procedures to the fact that she had encountered them in the one-credit class, Amy indicated that the procedure topic was challenging her as prior topics had not. Alluding to the BQ assignment, she described having a local variable that should have appeared in the parameter list. She elaborated:

> I think it's difficult just knowing what goes where, because you can have five procedures and the same word meaning different things. All the different variables and parameters that mean the same thing or mean different things. For example, for Number, you can use Num1, you can use Number, you can use Num. Just keeping them all organized, knowing what you are going to use for which one, and where it goes. I think just organization.

Amy emphasized organization, but she seemed to be echoing the literature position regarding the information-processing demands on working memory. Additionally, although the parameter instruction was essentially complete, Amy was still unsure about some terminology. For example, she substituted the phrase "the parameter that is in the main" for the term "actual parameter." Others, including Greene and Collier, have commented on the surfeit of additional terminology associated with the parameter concept.

Amy elaborated on the difficulties she was experiencing. "I think the things that really confuse me are the value and the VAR parameters and the local variables, just knowing when to use them," she admitted. When asked what helped her to make those decisions, Amy replied:

> When I do my programs now, I seriously think of a piece of paper or a pile of paper. It helped a lot. When I do my program, I seriously think, "Do I want this piece of paper back or the pad of paper back?" I mean it helps me in that way.

Once more, the value of concrete representations became apparent.

Of a value parameter, Amy said, "It doesn't come back out." Her conception of variable parameters included the observation that changes to variable parameters are reflected in the original variable. Amy's terminology consistently reflected the communication view of the parameter process. Moreover, her words, "get a number back that can now be placed in the original variable," implied a replacement of the actual parameter at the conclusion of the procedure's execution.

Analogy

The analogy portion of the interview provided the first opportunity for Amy to apply her understanding of the parameter process. When posed with the Mona Lisa analogy, Amy offered that she would consider situation two similar to a variable parameter because

> they're sending the painting to somebody else. That somebody else would be the procedure doing what needs to be done to adjust the painting. Then they're sending back the painting.

Amy thought situation one would be classified as a value parameter. She justified her response, saying, "Because they're keeping it. They're sending back a different one. That would be a variable parameter, like painting two would be a variable parameter." Not surprisingly, since Amy's expressions consistently confirmed her adoption of the communication outlook, the movements of the paintings guided Amy's thinking. She connected the Mona Lisa model, which was retained, with a value parameter and the student paintings that were "sent back" with variable parameters. In terms of her understanding, Amy analogized correctly. A modified-and-returned painting represented the variable parameter, and a painting that was not returned represented the value parameter.

Task 1

After displaying some initial confusion regarding the structure of the program--possibly due to nervousness generated by the interview situation--Amy began analyzing the code. Her observation that "N1 would be I, and N2 is J. N1 is 16" indicates that she understood the actual-formal parameter connection. Beginning to trace the code within the procedure, she correctly observed "T would be 16; then N2 would be 16 also." Amy's partial trace was correct; it was, however, incomplete. Hence, it appears that her initial effort concluded with 16 in all three procedure variables.

Amy next asked if it was possible that there could be more than one correct heading line. If her original trace did place 16 in all three variables, then three of the four procedure lines would have produced the desired output, in which case Amy's question was germane. In response to her question, I (the interviewer) offered that there were four possible combinations of two numbers and four procedure heading lines. The response to Amy's question may have been inappropriate, as it could only have been interpreted correctly if the procedure swapped the values (as its name implied). Although Amy had, during her initial efforts, alluded to "switching the numbers around," she had leapt into the details of the task and was even initially confused about the program's structure. It is possible she was still in the process of formulating for herself the outcome of the procedure statements.

In response to the observation about the possible answers, Amy replied, "I still think it's number three because it puts the value of N1 which is equal to I which equals 16 into T. Then N2 equals N1. N2 is 5, so N1 would now equal 5," as she correctly traced through the statements of the procedure. This time she did not conclude that all the values were 16. Moreover, she acknowledged that she recognized the code as the Flipper program. She responded correctly (5 16) when asked what the output would be if the procedure worked correctly, and followed up by indicating she did not want line three any more "because the same number's coming out. One of them has to be variable and one of them has to be a value." Prompted with a question regarding the program's output if procedure line one was the correct choice, Amy responded by orally working through the procedure's actions and selecting procedure line one, the correct choice. Considering that the abstract task may have initially been confusing to the participant, it is possible that the reference to the outputs assisted in her ability to focus on the procedure heading lines. It is equally plausible that she understood the effect of the procedure heading lines and was instead initially confused about the effect of the procedure statements. Asking Amy what the output would be if procedure line one was the choice may also have encouraged her to consider procedure heading lines other than the familiar procedure line three, the one that would have produced the output she knew a correctly functioning procedure should have. Whatever provided the link, it appears that once Amy was able to relate the problem to her previous experience--to recognize it as an incorrectly structured Flipper module--she was able to solve it quickly. As Amy missed none of the procedure-related questions on the test two days later, one could speculate that Amy profited from the opportunity to elaborate her conceptions of the parameter construct.

Task 2

The third interview, conducted November 18, began with the paper-and-pencil construction task. As Amy examined the procedures in the order of their appearance on the page rather than the order of their execution within the program, it was apparently the concrete, physical representation of the program rather than the logical representation that guided Amy's efforts. She correctly chose value and variable parameters appropriately, saying, for example, "It's a value parameter because it doesn't need to come back to the main. It can just stay up there." She provided correct formal parameter names, except that she chose the parameter name Volume instead of Answer in the DisplayOutput procedure, indicating that she chose Volume because she used that name in the CalculateVolume procedure. As she correctly identified all other parameter names and used different actual and formal parameter names in the BQ assignment, it is likely that her mistake was no more than an oversight.

Task 3

Amy began the modularization task by stating that she would first cut and paste the user instructions to make a procedure for that purpose. Then, referring to the paper copy of the program statements, Amy looked at the details of the defined constants (InputPrompt2 and InputPrompt3), the statements that physically appeared first in the program, and contemplated constructing procedures for them. Only then did she stop to question the function of the program. Upon receiving an explanation, she decided to construct a multi-purpose module, which she named CalculateGrossPay, to obtain the necessary input and calculate the gross pay. That completed, Amy inquired, "Is this program only going to go through once?" Again, it appeared that she was working from the "bottom-up," attending first to the details and later to the over-all program structure.

Next Amy constructed an output procedure. She took care to use different actual and formal parameter names, saying she did not want to confuse them. Because she had neglected to change a variable name as she moved a statement into a procedure, her first compilation attempt failed. Her next attempt compiled successfully; however the execution produced zero values for HoursWorked and PayRate because Amy used value parameters instead of variable. As GrossPay was correctly identified as a variable parameter, it may be that she was relying on her prior experience with single-purpose calculate modules, in which case parameters comparable to HoursWorked and PayRate would appropriately have been value parameters. Amy made the necessary corrections without hesitation; she apparently recognized the symptom and knew the treatment.

One explanation for Amy's actions posits that they simply reflected her preferred cognitive style. Cheny (1980) reported that "the heuristic approach is characterized by a trial-and-error method and the use of feedback to adjust the course of action chosen" (p. 285) The description seems to fit Amy's behavior.

Alternatively, Amy's actions could be interpreted to suggest that she was relying on the computer to assist her problem-solving efforts. If this is so, the "electronic coach" may have been acting as the more able peer, helping her to complete assignments successfully (Pea, 1985). On the other hand, the computer-provided assistance may have relieved Amy of the need for "mindful abstraction" (Van Merrienboer & Paas, 1990; Van Merrienboer & De Crook, 1992). As long as the problems were routine and the programs produced the correct answer, she did not have to ponder seriously whether a parameter should be value or variable.

Task 4

Amy undertook the code tracing exercise and proceeded without hesitation. For this task her approach was more structured, as she looked first at the program's structure. She verbalized her understanding as follows: "The main program has N equal 3. Then it calls a procedure, putting 3 in two different places; it does N twice." Amy's last statement calls attention to a misconception that can be attributed to her total reliance on the communication perspective of parameters. As the communication view provides no image of the manner in which the two-way communication is accomplished, learners are forced to construct their own model, a model that, according to Mayer (1982, 1985), may be impoverished. The process, in which two copies of the 3 were made, that Amy was attributing to the variable parameter was in fact the value parameter process.

Amy continued, correctly tracing the action of the procedure statements, arrived at a value of 4 for N "because you changed the value of N when you went through the procedure." Considering that Amy described a variable parameter with the following statement, "It has information that will be sent to the matching variable up in the procedure itself. Then the information can change, and you get a number back that will now be placed into that original variable," she answered correctly in terms of her understanding. She recognized that changes made to A and B changed N, and she knew that the change happened twice. Her answer was reasonable, given the firm entrenchment of the communication outlook.

ReadLn

Amy offered that ReadLn would have variable parameters because, in her words, "When you enter a number, more than likely you would use the number somewhere again in the program, other than having a ReadLn and output right after that." She then concluded that the ReadLn would have value parameters "because when we put it in, we can't change it until we call it again. So if I put ReadLn and I entered a number in, I couldn't change the value unless I called it again." Amy's mental model of the parameter process, one that strongly reflected only the communication viewpoint of parameters, did not enable her to visualize the behavior of the abstract and unseen ReadLn procedure.

Case Study: Sammy

A Capsule of Sammy

Sammy used the terminology of the text and classroom instruction easily, perhaps due to the reading she reported doing. In addition, she could construct generally correct programs. Her BQ assignment used identical actual and formal parameter names, but the later interview construction task did not. During construction tasks, she usually chose value and variable parameters correctly, giving appropriate reasons, with the exception that she sometimes used variable parameters in the output procedures.

Sammy struggled with the first analysis task. As she observed that assigning values to value parameters made no sense, it may be that the entire abstract task made no sense to her. As she continued her problem-solving efforts, two naive conceptions about value parameters emerged. She was uncertain whether or not value parameters returned values when the actual and formal parameter names differed, and she also did not know if changed value parameters returned values to the calling module.

Encouraged by the interview probes to verbalize her thoughts, Sammy worked through her fragile understanding and eventually solved the problem. She attributed variable parameters to the ReadLn, but then expressed confusion when asked to imagine the behavior of a ReadLn with value parameters. Sammy's first answer (4) to the last analysis task reflected the prevailing communication view, but she did express a small amount of dissonance. Sammy, the contextual thinker, repeatedly revealed that she learned from concrete models. Although she several times used the term "copy" in conjunction with value parameters, her own words divulged that she had formed no mental picture of a value parameter. To hypothesize that Sammy would have benefited from additional exposure to the more concrete, pictorial copy versus shared memory-location view of parameters seems valid.

Background

Sammy is a female under the age of 21. A junior, she is an English major with CIS and technical writing minors. Her transcript shows a B/B+ grade-point average; it also discloses that she earned a B- in her first semester, and that the overall grade-point has improved consistently since then. Indicating she had used a computer extensively both as a high school yearbook editor and during several summers of employment, she described herself as mainly self-taught. She took the one-credit Pascal course in the semester preceding her enrollment in CIS 110, and earned a grade of A-. The earlier course Sammy described as "neither easy nor difficult, just there." At the semester's outset Sammy indicated that she was confident that she would succeed in CIS 110; she earned a B+.

Referring to her disposition towards computers, Sammy described herself as excited because she realized how much there is to learn. She said:

> I have so much learning to do with computers. It's kind of neat because you learn something, and it's almost like the first rung on that ladder that everybody's got to climb. It's kind of neat because you know there's a lot more you can learn.

Sammy's observations mirrored the Turkle and Papert (1990) description of contextual learners, people who view computer experiences as an interactive conversation, in contrast to analytical learners, who think in terms of controlling the machine.

Sammy is a non-formal thinker, earning a score of 13 on the IPDT. She missed question 49 (a shadows question), 52-56 (all four classes questions), 59 and 60 (distance questions), and 70-72 (three of the four probability questions).

Reactions to the Course

The November 9 interview began with a question about her reaction to the course thus far. Sammy confessed, "It's not a breeze. Challenging would be a good word." Asked what challenged her most, Sammy said the amount of work; however, she added that she also found the content challenging. Although it is not uncommon for programming students to complain of excessive work, Sammy's problems with the content likely exacerbated her perception that the course involved immoderate amounts of work. The students who were struggling rarely finished the laboratory activities in the two hours provided; those who were not usually did.

Asked to describe the most difficult elements of the course, Sammy answered, "It's unfamiliar. I don't have a whole lot of Pascal background. That's the main problem." Having completed the one-credit Pascal course, Sammy probably had more background than most students who enroll in the CIS 110 course (which has no prerequisites).

Asked about her least favorite assignment thus far, Sammy, without hesitation, identified an earlier gas meter problem. She attributed her difficulties to the mathematics involved in the program, observing, "I'm not a math person. I like playing with numbers; it's just that they don't agree with me." Sammy described all of her mathematics classes as the same except for geometry. Of that class she observed, "The teacher was real good. He'd draw everything on the board." Sammy's observations about her high school mathematics classes and the instruction from which she most benefited foretell her perceptions of the programming class.

Sammy named the BQ assignment as her favorite, despite the challenges it posed for her. She said, "I had to figure out what was going wrong with it, and then I couldn't," and described spending five unsuccessful hours working on it, re-reading the text, asking questions in class, and then spending five more hours before she completed the assignment. Her words testified to her persistence and her determination to succeed.

Continuing, Sammy commented that the laboratory exercise (Lab 7) had been "kind of easy" because it entailed only "procedure parameters and a call." Sammy's statements echoed the intention of the laboratory exercise: to provide an initial experience allowing students to focus only on passing parameters. Moreover, she recalled asking "a lot of whys" of her classmates during that laboratory period. Her statements reflected those of Greene and of the considerable literature regarding the benefits that accrue from collaborative learning. The structured laboratory environment, with the students seated side by side working on the same problems, facilitated desirable student-student interactions.

Although Sammy provided vivid details of her struggles to complete the assignments, they were correctly done except that the BQ assignment--the one she had just completed--

inappropriately used VAR parameters in one output procedure. The BQ assignment did, however, use identical names throughout for the actual and formal parameters. Sammy missed only three (out of approximately 25) parameter-related questions on the test two days later.

Parameters

Of parameters, Sammy observed, "It took me a while to get the hang of it." She indicated she had been confused about the terminology: "what to call the parameters on top, and what to call those in the call." Her statements added to the mounting evidence attesting to the increased complexity caused by the surfeit of new terminology.

Sammy concluded it was not only the new terminology, it was the new concepts that were giving her difficulties. Likening her current situation to a biology major thrown into the middle of a Shakespeare class, she observed, "He'd be lost. I mean he wouldn't know what you were talking about when you start talking about all the forms you can write in." Continuing, she astutely remarked:

> That's kind of like what it is right now. I'm being handed a new concept on a plate, and it just takes it a while to separate it all into its own category. I've never encountered it before. It doesn't make it impossible by any means, just challenging. I've never encountered anything like this before.

Claiming that reading and re-reading the text had been very helpful, Sammy employed the words of the authors. She also reported asking many questions in class, a fact verified by the classroom observations. Asked about helpful instruction, Sammy referred to Greene's summation. Then she alluded to in-class payroll program simulation, calling it an "adventurous activity." In her words: "The sheet of paper thing that helped a lot because now this is on a pad because I want to take it back, so it has to be variable parameter. I don't want it to stay there." Sammy's words spoke to the value of concrete representations of the abstract concepts, particularly those that are foreign to the learner's prior experience.

Sammy's concise description of value and variable parameters echoed the words of the textbook and precisely described the communication view of the process:

> *In* parameters can take information into a procedure, but can not take it back out. *In-out* parameters allow you to put it into the procedure and take it back out so you can use it in the main program or anywhere else you want.

Asked what happened if a procedure changed a value parameter, Sammy claimed not to know "because changing a value parameter made no sense." She perceptively observed, "I would think that if it is going to change, you would make it a variable parameter." Sammy again repeated the rule for value parameters, but then she added, "You would have to name it something else down there to get it, down there in the main program. I'd think you have to name it something else to get it back out." Sammy's response revealed a naive understanding of value parameters. Although she fluently repeated the words that described the actions of a value parameter, she was unsure of the effect. She thought perhaps a formal value parameter with a different name from the corresponding actual parameter might "bring the information back out." She concluded the interchange by once more correctly repeating the rules for value and variable parameter action.

Analogy

Presented with the Mona Lisa analogy, Sammy offered an expressive response:

> In the second one, it seems as if you have to use a variable parameter if you are going to do a procedure about it. You're taking the Mona Lisa into the restoration master with the procedure. You're taking it in to him, and you want to take it back out. You would need to have a variable parameter for that. Otherwise you couldn't take it back out. Otherwise the restoration master would have it, and he could keep it. There's no way of getting back otherwise. If you just make it a value, he has it for good.

Apparently, the painting's destiny was guiding Sammy's thinking. As she eloquently observed, if the painting were a value parameter, the restoration master would have kept it. Given the text and classroom treatment of value versus variable parameters, and her consistent use of the communication-view terminology, her analogy was apt. Sammy also correctly noted that the restoration master would have received a copy had it been a value parameter. Although she used the word "copy" when referring to a value parameter, her overall analogy firmly reflected the communication view.

Turning to the situation-one analogy, Sammy voiced her understanding by first observing that the painting would be a variable parameter because "the instructor returns it to the supply closet." Then she surmised it would be a value parameter "because it stays in the art room," while the copies the students took home would be the variable parameters "because they're taking them out." Once more, a painting's final destination determined whether it simulated a value or a variable parameter. In Sammy's mind, the painting that remained was a value parameter, and the paintings that moved to another location were variable parameters. Viewed from the communication (or movement) perspective, Sammy's interpretation was not unreasonable.

Task 1

Presented with the analysis task, Sammy immediately looked to the procedure lines. She commented that lines three and four were the same, looked closer, and corrected herself. Sammy's momentary hesitation over the two procedure heading lines suggested that the form of the parameter lists confused her.

Sammy quickly recognized the task as the Flipper program she had done during the previous laboratory session. She explained that "you have your triangle here, and that's [the three variables] what you need to flip the numbers around," as she easily traced through the actions of the flip. Acknowledging a compliment to her explanation, Sammy volunteered that she had originally seen the flip as a seesaw, that she could not take full credit for the idea. She offered the following:

> Amy was sitting behind me in the lab, and we were both trying to figure it out. She [Amy] said, "Well you'd kind of have to make it so you'd have three variables." I [Sammy] said, "So you mean a kind of a triangle." Amy said, "Yes, that's what I was thinking." So it kind of came as a collaborative idea. I can't take full credit for that one.

Sammy's observation again attested to the value of the collaboration promoted by the laboratory environment. The mutual validation the two students provided for each other reinforces the idea that students do learn from each other.

Once Sammy determined the procedure's behavior, she turned her attention to the procedure heading lines. She began by expressing her confusion that the variables were named I and J in the main, and N1 and N2 in the procedure. The different actual and formal parameter names confused Sammy briefly, perhaps because her BQ assignment used identical names throughout. Then she observed:

> You're calling it different things, which doesn't make a difference. Here's where it's kind of fuzzy for me. I know if you have variable parameters, you can use those same names down in the mainline program.

Given her earlier confusion regarding the effect of a value formal parameter with a different name than its corresponding actual parameter, it may be that the remark about being able to use either the same name or different names for variable parameters was significant. She perhaps did not know the effect of identical formal and actual value parameter names.

Recognizing the procedure statement behavior and the heading line that would cause the procedure to operate correctly, Sammy adroitly observed that the numbers would not swap if the heading line were other than the third one. When her attention was called to the output (16 16), Sammy confessed, "That's what I don't understand. It's actually not doing anything because N1 and N2 are both N1 (16). I don't understand how you do that." Sammy knew that variable parameters returned values to the calling module, and she knew that the procedure swapped the variables. If, however, she believed--as seems plausible from earlier statements--that value parameters whose formal parameter name differed from the actual parameter name also returned values to the calling module, then the reason for her confusion is clear. There was no possible way of producing the output the program was claimed to have. Irrespective of the procedure heading line, the output would have been 5 16.

Sammy once more expressed her confusion that the main module variables (I and J) were not incorporated into the procedure. However, her correct observation that I matched up with N1 in all four cases confirmed that she understood the actual-formal parameter connection. Sammy orally traced the statements several times more, and several times more expressed her confusion. Then she remarked, "I've never encountered this before, where you're assigning values to your parameters." Given the response, it is likely that Sammy was saying she was confused by the assignment of a value to a value parameter within the procedure body. Earlier in the interview she claimed she did not know what would happen if that was done. Interpreted in terms of the earlier statement, Sammy's confusion was understandable. She knew the parameters could not both be variable parameters, but she did not know what would happen if an assignment was made to a value parameter. Coupled with her uncertainty over the effect of value parameters with different actual and formal parameter names, confusion was the only possible state.

After more tracing and more expressions of frustration, Sammy again alluded to the Flipper program. She recalled the procedure heading line she had used there, and she knew the outcome: the values had "flip-flopped." Specifically, Sammy observed that if the values had gone in 16 and 5, they would have returned 5 and 16 had procedure line three been the correct choice. Sometime later, she again expressed her bewilderment:

> There's just something I'm not understanding. I'm trying to figure it out. If this one doesn't come out, and if it's not supposed to come out, then I don't

> think it counts in your call line. Yet I put a lot of things in my call line yesterday for that program that didn't count. The values didn't come out with it. They were just there.

As Sammy appeared stumped, I suggested that she try the process of elimination. Then, as she seemed discouraged, I offered her the option to stop. Sammy chose not to stop; however, she professed that she did not know what to eliminate. Reminded that she had already eliminated procedure line three, she immediately eliminated procedure line two "because all those values stay in the procedure." Asked what the output would look like if procedure line two were the correct choice, Sammy once more traced the swap, remarking that the swap seemed to work fine no matter how she did it. Asked if there was any reason other than the statement in the procedure body that might account for the variables not coming back swapped, Sammy responded, "The procedure line."

Referring to the fourth procedure heading line, Sammy correctly observed that the value parameter would not be "taken back out" and wondered, "I don't know if it still counts in your call line or not." As she had used the phrase "does not count in the call line" several times, I questioned the meaning. Apparently substituting the phrase "call line" for "procedure heading line," Sammy explained:

> Remember when you asked me before if a value parameter can be brought out of a procedure call line if it's called something else? I said I didn't know. It's the same thing with this one. I don't know. If that is possible, then that changes everything.

Sammy's statements confirmed her earlier assertions regarding her fragile conception of value parameters. Despite the fact that she accurately repeated the words describing the behavior of value parameters, she was uncertain about the effect of changes to value parameters. As Sammy aptly observed, "If that is possible, then that changes everything."

Sammy once more repeated the value and variable parameter rules, asserting, "Value is in, and variable is in-out." Asked what that meant in relation to value parameters, she answered, "They stay in the procedure; they can't leave it." Without hesitation, Sammy correctly traced the action controlled by procedure line four, but again expressed her frustration at her lack of understanding. Encouraged to continue her thinking, Sammy correctly traced the action still another time. This time, however, she explained her reasoning in a manner that demonstrated understanding. Moreover, she immediately identified the first procedure line as the appropriate choice, musing, "How come I didn't see that before?" Thereafter, Sammy unhesitatingly traced through all four procedure lines and justified her choices with appropriate explanations. She correctly observed that procedure line two was a useless procedure because it swapped the numbers, but did not return them, and concluded, "I think I'll go with door number one."

When Sammy began the code-tracing task she first appeared stymied, articulating her lack of understanding repeatedly. One reason may be that the task was abstract, with no connection to a concrete--or relevant--reality for her. It bore little resemblance to the more contextually rich problems Sammy had previously encountered in the laboratory exercises and the assignments. Furthermore, Sammy had several times said it "made no sense" to change a value parameter. It may be that the task made no sense to her in terms of her understanding of when a value parameter is appropriate. That would not be an inaccurate assessment of the task. Sammy could not reconcile the program's behavior with her past experience; it did not behave as she knew it should behave.

Confronted with this new and contradictory situation, she returned again and again to the familiar procedure body, attempting to explain what was happening in terms of her current understanding. Eventually, Sammy's continued problem-solving efforts revealed a fragile conception of value parameters, a conception she was able to consolidate as she explained her thoughts. Given the scaffolding and the support the interview situation afforded, Sammy persisted and eventually understood. Her statement "Why didn't I see that before?" recalls the flash of understanding that comes after periods of struggle. One cannot help observing the potency of verbalization, as noted by Mayer (1985) and others.

Task 2

The third interview, held on November 29, commenced with the hand-construction task. Sammy examined the procedures in the logical order of the program's execution. Her only mistake was making the DisplayOutput parameter Answer a variable parameter. Asked why, Sammy replied:

> I made it VAR because you have to take it back out of the program and put it in Volume when you display it. You have to take it out. If it's going to go in and stay in, but it's coming out. That doesn't tell me much.

Evidently, Sammy was still unsure about when to use a value parameter. She knew the rules of taking it in and "staying in" or "coming out." In her own words, though, the rules did not help her very much. One could speculate that the opaque, one-way versus two-way communication view of parameters was not helping Sammy to understand when a variable parameter was needed, or when a value parameter would suffice. She needed a concrete image of what was actually happening to guide her decision making.

Referring to the CalculateVolume procedure a short time later, Sammy said:

> It's just not making any sense to me. I guess I don't understand how these would be moved up there and used in there, but I don't care really. I've never understood how it got into the procedure. Never mind, I answered my own question. If it's already in these three [mainline variables], that's how it got up in there.

Sammy was expressing confusion over how the CalculateVolume procedure obtained access to the values it needed to calculate the volume. Although she rather quickly recognized that the GetInput procedure had acquired them, and seemed satisfied despite a cloudy understanding, it may be that Sammy would have profited from the more transparent, shared memory-location view of parameters that rendered explicit the manner in which the mainline variables were altered.

Task 3

Sammy next moved to the computer-assisted modification task. She looked first at the input prompts provided as user-defined constants, remarked that they were unnecessary, and retyped them in the body of the GetInput procedure. Sammy 's retyping efforts seemed to be an example of the contextual learner's need to stay in touch with the form of a program at all times during the construction process (Turkle & Papert, 1990). The user-defined constants added another level of abstraction, one that perhaps hampers the contextual learner who needs to stay in touch with the details and form of the program. Although students do need to incorporate user-defined constants in their programs, requirements for their inclusion may be inappropriate when the novices first encounter otherwise unrelated concepts.

Sammy availed herself of the hard copy listing of the program and took a moment to partition the program into the anticipated modules, an obvious instance of planning and using a concrete aid to assist in the planning. Sammy constructed the program correctly, except that once again the display procedure had variable parameters. She checked over the procedure calls and the parameter lists before compiling; her first attempt was successful.

Task 4

Concentrating, Sammy proceeded to trace the details of the program statements. Although she appeared perplexed and debated briefly, she decided the WriteLn would display a 4. Acknowledging that she chose 4 because A and B were variable parameters and the changes had to be "brought out," she added, "The fact that they were called Ns replaces this value then. So the WriteLn displays whatever was in N; 4 was in N." As Sammy realized that A and B replaced N, it may be that A and B having the same value allowed her to resolve the dissonance she was expressing. A better task would have been one that adjusted the parameters A and B with different values.

Asked how she imagined that changes to A and B changed N, Sammy replied:

> When you call your procedure, like here you're calling Increment, and you're calling A N and B N. It's kind of like a hand: once the procedure is done, it's kind of like a hand that's going out and grabbing whatever is in A and B and bringing it back down to the main line. The way I look at it is that whatever was in A is in N, whatever was in B is in N, and N is 4, so . . .

Although the picture Sammy's words paint is more vivid than those of some other participants, she shared with them all the naive conception of the formal variable parameter replacing the actual parameter at the procedure's conclusion. It seemed the order of the appearance in the formal parameter list governed the order in which Sammy's imagined "hand" replaced the Ns.

To a question about her image of a value parameter, Sammy answered:

> It just kind of is put in the procedure and stays there. It doesn't come back out even if it's changed. In order for you to bring it back out, you have to call it a variable procedure, and you have to indicate that by putting VAR before it.

Sammy's response confirmed that she had clarified her understanding of the effect of changes to a value parameter. Moreover, it affirmed once more her total reliance on the communication view of the parameter process.

ReadLn

Sammy offered that ReadLn's parameter needed to be variable. Asked why, she responded that nine times out of ten the program would be using the value elsewhere. When asked, "What if the ReadLn parameter were a value parameter instead of a variable parameter?" Sammy responded, "I have no idea. It would still be user input, though, because they're still inputting input/output. I don't know. Honestly, it doesn't make any sense to me." One can hypothesize that Sammy, who acknowledged having no mental image of how value parameters are passed and who relied on concrete models, was simply not able to picture the unseen parameter list of a ReadLn with value parameters.

Case Study: Sondra

A Capsule of Sondra

Sondra consistently constructed programs that produced the correct answer. However, in all instances where he was given a choice, he used identical formal and actual parameter names. Although he admitted not always being sure of his choices, Sondra chose value and variable parameters correctly, and provided appropriate reasons for doing so.

While claiming that he was unsure of the terminology, Sondra was able to repeat words that reflected the communication perspective of the parameter construct. He was, however, unable to apply that knowledge to correctly solve the first analysis task. Moreover, as Sondra described the unseen ReadLn, he once more correctly echoed the language, but was again unable to apply that knowledge. One explanation for Sondra's limited success with the analysis task and the ReadLn exercise is their abstract nature. Another is that his understanding was fragile. Additionally, Sondra harbored a misconception regarding the parameter syntax, which likely contributed to his limited success on the first analysis task, and probably hampered other work.

Of all eight participants, Sondra seemed closest to adopting the shared memory-location perspective of the parameter construct. He used the terms "copy" and "original" with apparent understanding, and he included the notion of data storage in his definition for variable. Moreover, his retreat from the communication-oriented answer in the second analysis task could be interpreted to mean it was in conflict with a budding sense of the shared memory-location perspective.

Sondra's score on the IPDT classified him as a non-formal thinker. His problem-solving approaches on the interview tasks--learning about a problem by examining its details, remaining in touch with the details, and employing trial and error--suggest a contextual learning style. In view of his apparent learning style, his reported lack of a mental image of the parameter process, and his emerging shared memory-location viewpoint, one can hypothesize that Sondra would have greatly profited from a transparent, concrete model that makes explicit the process by which parameters are passed.

Background

Sondra, a male under the age of 21, is a freshman majoring in paper science with no programming experience before enrolling in CIS 110. Apparently unaware that the course is a requirement in the paper science curriculum, Sondra elected the class because he lacked confidence in his computer fluency. He volunteered that he had some concerns at the semester's outset as to whether he really belonged in the class. He earned an A-, slightly better than the overall B average he earned during his first semester at State University.

Sondra is a non-formal thinker who earned a score of 17 on the IPDT. He missed questions 54-56 (three of the classes questions), 59 and 60 (two of the distance questions), and 71 and 72 (two of the probability questions).

Reactions to the Course

During the second interview, held on November 10, Sondra described the course thus far as "a challenge." He admitted that he was sometimes confused for "a day or two." He added, however, "I pick up on it after a while." Sondra identified the lab as the facet of the course he most liked. Referring to the bounded activities that allowed the students to focus solely on the parameter process, he said:

> I tend to like stuff that I understand and I can comprehend. Some of the labs--just our past lab--where we did a few parameter things [referring to Lab 7], that's when I was starting to understand parameters and what was going on.

Continuing, he said:

> I kind of liked it because I could understand what's going on. I could comprehend it. Sometimes I see something and think "Oh my gosh! What does this mean?" I kind of get confused, and I don't like this. It's just natural for me. If I understand something, I like it. If I don't understand something, I automatically don't until I do understand it. Then I say, "Oh, it isn't that bad."

Sondra's statements have implications for the disposition research question. For Sondra there was a vital relationship between understanding and a favorable disposition toward the computing discipline.

Sondra "kind of liked" the banner and the revised banner assignment, offering that it provided a chance to be creative. The BQ assignment he described as his least favorite because he "found it very difficult to do, keeping all your variables straight and in order." Sondra's version, although it displayed some design and style shortcomings, passed parameters correctly, albeit using identical actual and formal parameter names. The procedure laboratory exercises were also correctly done, although the Flipper program--in which Sondra was free to choose parameter names--also matched the actual and formal names. Sondra missed three procedure-related questions on the test that he took the day following his second interview.

Parameters

As Sondra had indicated he found the course challenging, a question about the relative difficulty of the topics seemed appropriate. Sondra offered that the constructs prior to parameters had not posed particular problems. He explained, "IF *this*, THEN *that* ELSE *it's going to be something else*. It is the same with the WHILE . . . DO: WHILE *this is happening, it's going to do that*. I mean the words explain it for you." In contrast, Sondra depicted parameters as challenging:

> It took me a pretty long time to catch on to those--about a week, perhaps a week and a half--before I actually really, fully understood what it meant if you defined a variable here, down there, and the order they had to be in. It took me quite a while to figure those out.

Sondra was describing one of the complexities of the parameter construct. In contrast to some of the other language features such as the WHILE and the IF statements, the Pascal syntax does little to assist the novice in developing a mental image of the parameter passing process.

Sondra elaborated on his confusion regarding the parameter process, singling out the idea of "what a value parameter could do, and what a variable parameter could do." He confessed that it took some time before he could understand the difference between the two concepts and when it was appropriate to use them.

Apparently the participant arrived at the point where the definitions became clear in his mind, for Sondra repeated definitions that suggested he understood the two concepts correctly. A value parameter he described as able to store information, but not send the information out of a procedure. Variable parameters, he said, can "put information into the procedure, and that information can be taken out." It is perhaps relevant that Sondra's words distinguished between the actions of the two different parameter types; they did not address the question of when it is appropriate to use them.

Analogy

The interview next turned to the Mona Lisa analogy. After some thinking aloud, Sondra offered the following explanation of situation one:

> The instructor brings a first class replica, so that means there's a place in the memory of the computer where you're taking the value out of there, but a copy of the value stays in there, stays in that memory location. You're bringing a copy of that out of the memory. After the students are done doing work with that copy, he's putting it right back there.

Sondra's explanation of the situation-one Mona Lisa reflected the idea of a copy of a memory location, one of the few references any of the participants made that directly implied the notion of data storage function of variables. Of the second situation, Sondra said:

> He's taking the original out of the memory location and doing work on it, but there's not a copy in the memory location because it's the original. He's taking everything out of the memory location, doing work on it, and putting everything back in.

Erring only in their implication of movement, Sondra's explanations, more than any offered by any other participant, embodied the shared memory-location view of a variable parameter. However, he incorrectly connected situation one with a variable parameter and situation two with a value parameter. Sondra concluded by speculating that he might have the two "mixed up for some reason."

Sondra did have the situations reversed. His error may have been no more than a slip, due to the strain of the interview situation. Conversely, his uncertainty at the end of the response suggests that he had not yet firmly connected the terminology and the concepts. Sondra's appropriate use of the terms "copy" and "original" as he analogized, as well as his allusion to memory locations, implies that he was constructing an image consonant with the shared memory-location view of the concept.

During the same interview, Sondra defined a variable as "a place in memory, and you are going to be able to store a value there." It may be that Sondra's image of a variable, an image that included reference to data storage, provided the scaffolding that sensitized him to the textbook and classroom references to the shared memory-location perspective. It may also be that Sondra, the contextual learner, could relate to the visual image of a memory location more easily than he was able to relate to the abstract rules of the communication perspective.

Task 1

Sondra next attempted to select the correct procedure heading line in the first analysis task. He began by going directly to the procedure's variable section, correctly observing that he could eliminate any procedure line that had a T in it because T was local to the procedure. Not finding a T in any of the procedure lines, Sondra commented he would need to look elsewhere for a connection. After several minutes of unproductive time examining the details of the procedure, he stopped to question the procedure's function, at which point he recognized the problem as similar to the previous laboratory exercise. Apparently unable to build on the prior knowledge, Sondra commented, "I'm looking at the last one and trying to decide if I need a variable parameter at all," and spent several more minutes tracing the procedure body's statements. After tracing the swap several times, Sondra tried to "plug in" another heading line "to see what happens." Apparently unable to imagine the procedure heading line's role in determining the output of the procedure, he returned to mechanically tracing the values through the statements in the body of the procedure.

Looking at procedure lines three and four, Sondra explained, "These two are the same thing. The only difference is this one has a comma." He was confused by the parameter syntax, incorrectly believing the parameter lists `(VAR N1, N2: Integer)` and `(VAR N1: Integer; N2: Integer)` to have the same effect. He did not understand that both parameters in the former are variable parameters, while N2 in the latter is a value parameter.

Referring to the triangle Greene had repeatedly used in the variable swapping demonstration, Sondra traced the code again. His statement that "N1 is going to equal 16 and N2 is going to equal 5" reveals that he understood the connection between the actual and formal parameters. This time he correctly identified procedure line one as the appropriate choice, correctly repeating words that described the behavior of the swap algorithm and of value and variable parameters. However, he seemed stymied by the fact that the body of the procedure correctly performed its task, yet returned incorrect values to the calling module. Finally, Sondra turned his attention to procedure line two, and after some time, eliminated it because there was no variable parameter, correctly observing that "there has to be one if they end up being the same thing."

During the interview, Sondra seemed to be constructing his understanding of value parameters, as is demonstrated by his eventual elimination of procedure line two. His misconception regarding the syntax of procedure line three may have camouflaged a better understanding of situations where all parameters are variable.

How might Sondra's problem-solving behavior be explained? His style mirrored the heuristic style described by Cheny (1980). Employing trial-and-error, Sondra "plugged in" one after another procedure heading line "just to see what happened." Moreover, Sondra, the contextual thinker, learned about the problem by examining its details. It may be that, needing to stay in touch with the details of the problem at all times, he could not envision the swap without repeating the details. Another possibility is that there was too large a gap between the abstract interview task and the Flipper program Sondra had written previously. Perhaps if the interview problem had been posed by asking him to identify the incorrectly constructed Flipper program that produced an output of 16 16, he would have been able to connect the task to his existing knowledge. Likewise, the additional abstraction of non-matching actual and formal parameter names likely increased the problem's complexity for Sondra, who consistently chose identical names.

Task 2

Two weeks later, on November 23, Sondra returned for the third interview, which commenced with the hand-construction task. After hearing the instructions, Sondra examined the bodies of the procedures to discover the procedures' tasks. As he considered the procedures in the order of their appearance on the page, Sondra was apparently guided by the physical representation of the program rather than the logical order in which the statements would be executed. He correctly assigned procedure names compatible with those used in the procedure calls. Having examined the details of the procedures, Sondra mused:

> I'm not sure where I want to start. For my CalculateVolume variables in my procedure, I'm just deciding what I want for my variables in that procedure line. I'm looking at the main program right now to decide what I want to put up there.

Sondra recognized without difficulty that GetInput needed a variable parameter "because the rest of the program needed that information to work." He correctly chose a value parameter for DisplayOutput although he admitted not being "100% sure" about the choice. Moreover, he correctly identified the CalculateVolume parameter that needed to be variable "because you need to get that information out." The remainder he made value parameters "because after you do the calculations you don't need those numbers any more." Before stopping, Sondra once more reviewed the formal and actual parameter lists to verify their accuracy; Sondra's version was correct. He did indicate, however, that he would have been more comfortable if he had a computer to check out his work and "see if it really works."

Sondra's self-reports suggest that he had not consolidated his understanding of the parameter construct. It may be that his reliance on the machine to "see if it really works" allowed Sondra to write programs that produced correct answers without confronting his fragile understanding of the constructs in question.

Task 3

Sondra started the modularization task by saying, "I'm just kind of looking at what it all has to do here," after which he immediately began building procedures and moving statements. In contrast to other students (who constructed a single input module or combined input and processing modules), Sondra built two input procedures, giving the two procedures names (InputPrompt2 and InputPrompt3) that duplicated the names of the defined constants rather than names that reflected the modules' purposes. He said:

> What I'm going to do with those constants, I'm going to make them procedures. I'm going to keep the same names. Right now I'm just making this into a procedure, and I'm just rearranging it so it can be done in procedures.

Sondra, the contextual learner, solved the new problem by rearranging familiar features of an existing one. Manipulating the familiar, he retained the input prompt names. Needing to stay in touch with the details of the problem, Sondra eliminated the defined constants and retyped the details of the WriteLns in the newly created procedures.

Sondra explained each statement in detail as he typed it, much the way he articulated each imagined move in the earlier analysis task. When asked why he had moved some statements while retyping others, he candidly admitted, "I never actually (small chuckle) looked down at the program. I didn't really look down in the program first before I did it." Again Sondra had learned about the problem by first examining its details.

The construction-task performance paralleled his class work. Given a choice, he chose identical formal and actual parameter names. He did comment however, that the names need not be the same; he just "found it easier." As Sondra correctly completed several tasks that required different actual and formal parameter names (including the previous hand-construction task), his evaluation seems likely. Turkle and Papert (1990) reported that some bricoleurs in their studies avoided sub-procedures, preferring to retype statements because doing so allowed them to stay in touch with the program's details. It may be that maintaining identical formal and actual parameter names helped Sondra to stay in touch with the details of the program. It may also be that using identical names alleviated his difficulty "keeping all the variables in order."

Task 4

Attempting the second analysis task, Sondra started by examining the details of the procedure. "I'm looking at the procedure just to see what this program is going to do. A equals A+1, and B equals B+1. A and B are both variable parameters," he correctly observed. Repeating each program statement, Sondra continued:

> The procedure call inside the program says (N, N). That means they go back up to the procedure. That means A is going to equal 3, and B is going to have a value of 3. So A+1, in other words, 3 + 1 equals 4. So A is going to equal 4, and B + 1, which is 3. So B is going to have the value 4. Then the procedure ends. My guess is that it's going to write 4 for our answer for WriteLn (N).

At this point Sondra paused, appearing a bit perplexed. "It seems too simple; that's the problem. I'm just looking at it, and there must be more to it than just this," he confessed. Articulating the problem again, he concluded, "A and B are going to have the value of 4. However, it just says WriteLn (N). I have two values, A and B, and it just says WriteLn (N)." Acknowledging that N was the main module's only variable, Sondra remarked:

> It would WriteLn a 3; it would display 3. Actually the procedure here means nothing because it doesn't WriteLn (A, B) in a procedure line. It just WriteLns (N) in the main program. N equals 3 in the main program, so it would WriteLn a 3.

His immediate answer of 4 was quite reasonable from the communication outlook, demonstrating his recognition of the fact that variable parameters return changes to the calling module. The revised answer (3)--given after a moment's reflection--suggested that he had abandoned the communication view. One interpretation might be that he revised his answer because the reasonable answer, given the communication perspective, conflicted with his budding shared memory-location perspective. During the Mona Lisa analogy, Sondra had referred to the variable parameter as the "original" variable. He had earlier defined a variable as a "spot in memory," and presumably knew that a memory location can hold only one value at a time. From this it is possible to conjecture that he held some image of a variable. On the other hand, Sondra denied having a mental image of the process that caused value and variable parameters to behave as they do. In the preceding task, the student was confronted with both A and B equaling N. It may be that his understanding of variables conflicted with his fragile knowledge of parameters. Without a concrete parameter model, one that was compatible with his understanding of variables, his knowledge of variables prevailed. Sondra abandoned what he knew about variable parameters--that they return values to the calling module--and decided the procedure did nothing. Conversely, Sondra's comment that "it was too simple" may mean that he suspected a trick, and responded with what seemed to him the only possible answer other than 4.

During the ensuing interchange, Sondra correctly identified the parameters as variable, and attested again that changes to variable parameters would be reflected in the calling module. Nevertheless, he repeated his statement: "All it says is WriteLn (N). What you're WriteLn-ing is N from the main program," and again confirmed that N retained the value (3) given by the main module. As Sondra affirmed that changes to a value parameter would not "come back," it would have been informative to query him what the output would have been if the parameters had been value.

ReadLn

The interview concluded with the question about ReadLn's parameters. Sondra correctly responded that they would be variable parameters:

> . . . because when you ReadLn, you're ReadLn-ing into that spot in memory. You need that piece of information later in the program that's using it. You're going to have to take it out of that place in memory again in order to use it.

His response mirrored the phrases he had used during earlier tasks when explaining his value-variable choices. "If it were just a value, you'd just be ReadLn-ing it in," he continued, "You really couldn't do much with it." The following query prompted Sondra to apply his understanding:

> If ReadLn had a value parameter, could the program do anything with it? [Referring to task 4] Instead of assigning a value of 3, what if it said ReadLn (N) and the person typed in a 3? If it were a value parameter, what would be in N?

He was unable to do so, and replied "3," the appropriate answer for a variable parameter. Asking Sondra to apply his understanding of a ReadLn with variable parameters would likely have been productive. Without further data, however, it is possible only to posit that Sondra, the contextual learner, could not visualize the abstract and unseen ReadLn parameter list.

Case Study: Steve

A Capsule of Steve

Steve fluently repeated words that reflected the communication perspective of the parameter construct. He easily accomplished the first analysis task, he correctly recognized that ReadLn has variable parameters and a ReadLn with value parameters could not return information, and he gave a reasonable--albeit communication-oriented--answer (4) in the last analysis task.

Conversely, Steve struggled with the interview construction tasks; his remarks suggested that he struggled with the class work also. Repeating several times that it was important to do so, Steve chose different actual and formal parameter names during the interview tasks. However, when he was required to construct parameter lists for existing procedures (where the parameter name is fixed), he did not consistently choose names compatible with the procedure statements. Steve also acknowledged that he sometimes used variable parameters "just to be on the safe side," implying that he was not always certain when value and variable parameters are appropriate. There is also some question whether Steve understood what happens when a variable parameter is used.

Steve's score on the IPDT classified him as a concrete thinker; moreover, he frequently displayed problem-solving behaviors characteristic of Turkle and Papert's (1990) bricoleurs. Furthermore, Steve's detailed description of the instructor's concrete examples attested that he learns from concrete models. Given the nature of his hypothesized misconception and his apparent preferred learning style, it is likely that

Steve would have benefited from additional exposure to the concrete, transparent model of parameters espoused by the shared memory-location philosophy of the process.

Background

A sophomore, Steve is a male under age 21. At the beginning of the semester, Steve listed CIS as his major and business as his minor. A former wildlife major, Steve had changed majors the previous semester while he was enrolled in the one-credit Pascal class. Returning to school following his father's mid-semester funeral, Steve fell behind in his biology class, a class required for the wildlife major. As he was "carrying a high mark" in the CIS class, he changed to a CIS major. Steve described the earlier Pascal course as fairly easy; he earned a B. He did acknowledge, however, having difficulties with several of the concepts covered in the course. Remarking that he really enjoyed the word-processing/spreadsheet class he had taken his first semester at State University, Steve volunteered that he had earned an A in it.

Steve is the only participant who spoke of grades. His academic transcript reveals his grades in the early CIS courses were consistently higher than his grades in other courses. From this information, it is possible to hypothesize that Steve developed positive feelings toward a course when he earned good grades. If Steve's sentiment prevails among students, then Dr. Greene's grading system (which was generous) would seem to support the development of a positive disposition toward the subject. The question would be a fruitful one to pursue, especially in terms of students' eventual success in the CIS major.

Apparently unaware of the demands of a college curriculum when he arrived at State University, Steve's academic transcript disclosed a 1.4 grade-point average during his first semester, an average that has since improved consistently. At the semester's beginning, Steve indicated confidence in his ultimate success in CIS 110; he earned an A-.

Steve is a non-formal thinker. He earned a score of 17 on the IPDT, missing questions 49 (a shadows question), 53-55 (three classes questions), 60 (a distance question), and 61 and 64 (inclusion questions).

Reactions to the Course

Midway through the semester, Steve described the class thus: "From challenging to exciting. If it gets too challenging, then you get frustrated. To get done and to know that your program runs and works, however, is a good feeling." Steve's words indicated that he equated a successful program with one that produced the correct answer. Observing that the course "started out pretty easy," Steve added, "once we hit procedures, though, it was tough."

Steve offered that his favorite assignment was the first procedure laboratory exercise (Lab 7). He summarized his reaction to the assignment as follows:

> We had three programs that were already typed up, and we had to go through and change certain parts. We worked with one other person, and my partner and I did really well on that. It felt good to just go through it and know what you were doing and complete it. That was one of the first times where it all sort of came together. I'd say it was my favorite just because of the way it challenged you, but it didn't stump you.

Steve's reactions attest to the value of the narrowly-focused laboratory exercises and the supportive, collaborative environment of the structured laboratory. He volunteered that finishing the laboratory exercise early had permitted him to begin working on the BQ assignment, which he described as his least favorite. "That was probably my least favorite. I just had so many problems and so many questions," he admitted. Commenting that he had worked on the assignment for about five hours, he recalled that it "was more thinking than typing. It had to be thought out." Despite his reported struggles, Steve's BQ assignment was well done and displayed excellent form. The only parameter-related error was one instance of a variable parameter that should have been a value parameter. It did, however, use identical actual and formal parameter names throughout. Lab 7 was correctly done; Lab 9 was also, with the exception of one inappropriately VARed parameter. A review of Steve's November 11 test, however, disclosed numerous parameter-related errors, with few other incorrect answers.

Parameters

Steve reported having difficulty with the syntax of the procedure heading line. His comment, "If you have a comma here, don't have a comma here, and they only identify up to a declaration of a variable" suggests that he would have benefited from exercises focusing on the different syntactical forms a formal parameter list can assume.

Speaking of parameters specifically, Steve commented, "The value parameters, they're no problem. It's just the variable parameters I have trouble with." He elaborated:

> I think it's the fact that you can have VAR sections within the parameters, or within the procedures. When you call them up, it's a different variable

> than it is in the procedure that you're calling. I have trouble grasping how to write out the procedure header and the procedure call. I think that the reason it becomes a problem is that it uses different forms of the variable that aren't the same.

Further probing seemed to suggest that Steve was having difficulty connecting the formal and actual parameters when the names are different.

Steve correctly noted that a "value parameter is passed from the main line to the procedure only, and not from the procedure back to the main line, so whatever is stored in the main line stays the same." His answer echoed his observation that value parameters did not pose a problem. He correctly indicated that a variable parameter "changes the actual parameter in the main line if it's changed in the procedure, because they can send it back and forth." When asked, "Do you have image of why a value parameter is not changed back in the main module and a variable parameter is?" Steve responded:

> If you don't declare it as a variable parameter, it's not sent back to the main line. It's kept within the procedure. I really don't understand why it isn't sent back or why it can't be sent back. Now I think of Dr. Greene with pieces of paper, saying, "If you just get a piece of paper, you can't send it back, but if you get a notepad you can." I really truthfully on the syntax level do not know why it doesn't work or why it doesn't send it back. I really couldn't tell you.

Steve's words affirmed the power of the concrete, in-class simulation. His expressions of non-understanding also provided an early clue to his apparent intuitive conception of variable parameters.

Analogy

Unlike other participants, Steve did not immediately relate the Mona Lisa analogy to parameters. Instead, the analogy reminded Steve of Greene's variable swap demonstration. He expressed his perception as follows:

> The first thing that came to mind was when Professor Greene brought over those sheets of paper, and he talked about how a copy was made, how you have some information in one area, and you copy it into another variable. Then it can be changed and copied back into that variable. That's the first thing that came to mind with the Mona Lisa, him talking about having to get his high-tech paper.

His observation once more attested to the power of concrete, everyday examples in helping novices to understand abstract concepts.

Relating the Mona Lisa analogy to parameters, Steve reported his belief that the situation-one Mona Lisa would be a value parameter because the "original Mona Lisa never leaves the possession of the instructor, and copies are made and taken home." Given the communication perspective, his interpretation is not unreasonable. Steve related situation two to a variable parameter because, in his words:

> The original is passed. For example, if the main line would be the owner, it's passed from the main line to the procedure so the restoration procedure can be carried out on the Mona Lisa. Then it is delivered back to the owner, or the main line. So there's a passing back and forth.

From the communication perspective, Steve's situation-two analogy reflects a variable parameter precisely.

Task 1

Steve accomplished the first analysis task (selecting the correct procedure heading line) rather easily. He looked first at the procedure statements, then offered, "I'm trying to figure out which one of them passes back and forth." He concluded by correctly tracing the swap code and choosing the proper heading line. Explaining his reasoning, Steve volunteered that he had seen the algorithm before, in the Flipper program. He observed, "I'm looking at the output. Instead of being swapped, they're the same, which tells you if this procedure was designed to swap, it doesn't work. The output should be 5 and 16." Evidently Steve, the contextual learner, provided his own context to the abstract problem, connecting it to a Flipper program that did not work. He ruled out the procedure line with both value parameters, providing an appropriate reason. He also acknowledged seeing similar code on the test, admitting that it caused him problems on the test. He said, however,

> Now that I've done this and talked through it, I understand it better. I understood it well. I can't say that I was really sure of myself, but talking it through, explaining it, and thinking it through certainly is helping me to understand.

Steve's experience adds to the mounting evidence supporting the constructivist position regarding the importance of explaining one's thoughts.

Without devaluing the ease with which Steve accomplished the preceding task, it is significant that his second interview took place the day following Dr. Greene's after-test review (during which Greene spent a considerable portion of the class period reviewing the code-tracing questions). In contrast, most of the other participants' second interviews either preceded or coincided with the test. Steve (like Carl, whose second interview was also late) fared well on the interview task, but poorly on the test. One can speculate that, for those students whose interview preceded the test, the interview itself provided an inadvertent instructional intervention.

Task 2

Steve began the two-part interview, conducted on November 30 and December 1, with the hand-construction task. He first went through the procedures in order of their appearance on the page, assigning procedure names of his choice. Then, commenting that he had just looked at the main module, Steve correctly changed the procedure names to match the procedure calls.

Attempting to construct the formal parameter list for GetInput, Steve mused, "I'm pondering whether these are local variables. I just don't understand. I'm confused by the three separate calls of the same procedure." He recognized that the input procedure needed a variable parameter "because it [the value] has to send it to `CalculateVolume (Length, Width, Depth, Volume)`. There's one thing, I know," he continued, "I know it reads them in the same order as they're entered in here because it needs to go to CalculateVolume." Steve's reference to the appearance of the two procedures' parameter lists (that the parameters needed to be in the same order) offered another clue to his imagined perception of a direct procedure-to-procedure link.

As the GetInput procedure seemed to be confusing Steve, I (the interviewer) suggested looking at the other procedures first. Doing so may have been a mistake. Because Steve was examining the procedures in the logical order of execution, the outcome of his efforts on the task may have been different had he continued with that approach.

Steve next worked on the parameter list of CalculateVolume. The procedure heading line (which follows)

```
PROCEDURE CalculateVolume (     Num1 : Real;
                                Num2 : Real;
                                Num3 : Real;
                           VAR  Num4 : Real);
```

he constructed contained the correct number of parameters, and the value and variable choices were correct. However, Steve selected the parameters names Num1, Num2, Num3, and Num4, none of which matched the names used in the procedure's only statement (`Volume := Len * Wid * Dep`). Steve offered that he had not chosen the names for any particular reason, that "the actual parameter list just needs something to match up with." Num4, he offered, needed to be a variable parameter because "it has to get passed between procedures to DisplayOutput after it's calculated." His phrase "passed between procedures" again suggested the direct link.

Steve next inspected the DisplayOutput parameter list (which follows).

```
PROCEDURE DisplayOutput(     Answer: Real)
```

He correctly named the single value parameter Answer "because the WriteLn calls it up as Answer." However, he also changed the Num4 parameter in the CalculateVolume procedure to Answer "because Volume is associated with Answer, but you can not have the same name in the main line and in the procedure." The revised procedure heading line follows.

```
PROCEDURE CalculateVolume (     Num1 : Real;
                                Num2 : Real;
                                Num3 : Real;
                            VAR Answer: Real);
```

Steve allowed that the program would compile if the actual and formal parameter names were the same, but said it would confuse the reader of the program.

As Steve had correctly chosen the identifier Answer because of its use in the DisplayOutput procedure's WriteLn, he apparently knew that variable names must be consistent within a procedure. However, Greene had strongly stressed the importance of using different names for the actual and formal parameters, an admonition that apparently impressed Steve powerfully. As written, the CalculateVolume procedure demanded that the formal parameter be named Volume, a requirement that conflicted with Greene's counsel regarding parameter names. One could conjecture that when Steve encountered a conflict between Greene's caution and the consistency requirement for variable names within a procedure, the admonition prevailed. However, other than possibly focusing Steve's attention on choosing formal parameter names that differ from the actual parameter names, and away from the need for consistency within the procedure, the explanation does not account for his choices of Num1, Num2, and Num3 in the CalculateVolume procedure. Alternatively, it may be that the added level of abstraction imposed by the requirement of different actual and formal parameter names hampered

Steve's construction efforts. As his correctly done BQ assignment had used identical names throughout, the interpretation seems plausible. The combination of evidence, however, will suggest a more cogent interpretation for Steve's construction.

The other procedure heading lines completed, Steve returned to GetInput. He once more alleged that the parameters needed to be VARs because "they have to go to other procedures." After an extended pause, and an offer to stop that he declined, Steve said:

> I don't know why it wouldn't work the same. If it were just regular, as if they were all in one line [In contrast to the input procedure in this program that was called several times, most input procedures Steve had seen obtained all the necessary input for the program during a single call.] But it's a new call each time. This is basically a part that I am having trouble dealing with.

It appears that Steve began by imagining a correctly constructed GetInput parameter list (which follows).

```
PROCEDURE GetInput (VAR Dimension: Real);
```

He admitted that this portion of the task was "kind of confusing," and correctly speculated that when the procedure GetInput (Depth) is called, it will "go through and run the procedure and associate Dimension with Depth." Then he questioned:

> But then when you call GetInput (Width), is it going to know to run the procedure and instead of putting it in Depth again, will it put it in the next one?

In answer to his query, I posed a general question about how the association was made, expecting an answer involving the actual and formal parameters. Instead Steve replied, "By sequence, normally, but usually when you see them with this . . ." If he meant to say that the connection is made by sequence within the actual and formal parameter lists, his answer was accurate. Then Steve mused to himself, "I don't know why it wouldn't work the same," and inquired, "I wouldn't have to put them in each time after this one, would I?" Assuming (an assumption that in all likelihood was faulty) that he was asking whether he needed a parameter list, I inquired about his understanding of the scope (the part of the program where the parameter is defined) of a formal parameter. Steve answered, "I would assume only that procedure."

It may be that the intended-to-be general responses to his parameter-related questions misled him into constructing the input procedure as he did. If that interpretation is correct, then Steve's understanding was merely fragile or incomplete.

On the other hand, it seems possible that Steve inferred the sequence interchange to mean either that the procedure would use a different formal parameter in each successive call, or that the procedure must have a formal parameter available for each unique variable used in a procedure call (that is, Num1 is associated with Depth, Num2 with Width, and Num3 with Length). Since there are three procedure calls and three parameters, either interpretation is possible.

It also seems that Steve may have interpreted the scope interchange in terms of the execution, rather than the compilation of a procedure. This conjecture, combined with Steve's earlier statements about not understanding how the procedure header and the procedure call should be written, and his lack of understanding the process on the "syntactical level" could mean that he did not understand the connection between an actual parameter and a formal variable parameter. He knew that a variable parameter returns a value, but it may be that he was unsure where it went. For example, Steve had earlier said: "A value parameter passes only from the main line to the procedure, and a variable parameter can be passed between procedures and back and forth between the main line and procedures." Speaking of changes to variable parameters he had also offered, "Then it's changed completely. If you pass a variable from a procedure to another procedure and that procedure changes it, then that is the value that's stored in memory." Steve's solution of the construction task provided evidence that he harbored a fundamental misconception of the parameter passing process.

Steve did not recognize the GetInput procedure as conceptually identical to the GetPercent procedure in one of the Lab 7 programs. He constructed the GetInput to match what he was more accustomed to seeing--one call to a procedure that secured all the inputs the program needed--naming the variable parameters Num1, Num2, and Num3. The provided procedure calls and the GetInput procedure heading line that Steve constructed to match the calls follow.

```
GetInput (Depth);
GetInput (Width);
GetInput (Length);

PROCEDURE GetInput (VAR Num1: Real;
                    VAR Num2: Real;
                    VAR Num3: Real;)
```

As Steve constructed numerous correctly functioning procedures over the course of the semester, one must assume he knew the number of actual and formal parameters must match. However, the conflict provoked by the seemingly novel input procedure caused Steve to abandon or distort the procedure-line construction rules (rules whose purpose he perhaps did not understand) to allow him to build a procedure heading line consistent with his understanding. After nearly an hour, the interview was suspended with the task in the state described.

Steve asked to return to the task when he arrived to continue the interview the next day. Recognizing that the GetInput procedure needed a declaration to match the `ReadLn (Dimension)` statement, he added Dimension as a local variable. Apparently referring to the GetInput procedure, Steve claimed, "That's the only one that is going to use the Dimension in that form because it's not used anywhere throughout, so it's not going to be put in the VAR parameter. It's a local variable." It appears Steve was imagining a direct procedure-to-procedure connection between the parameters, a connection that was established through identical parameter names. As no other procedure had a parameter named Dimension, he assumed that GetInput did not need one either.

Task 3

Next Steve undertook the modification task. After examining the program briefly and creating a simple procedure for user instructions, Steve began constructing what would become the input procedure. He typed the input prompts, saying "Now we're going to get rid of the constant section." Steve perused, but did not modify, the paper copy of the program listing, commenting, "Just so as I change this I don't get confused about what it was originally supposed to do." Using techniques that allowed him to remain in contact with the program details, Steve solved the problem. As he finished the GetInput procedure, he commented that "There is always revising to be done with me." As with the Turkle and

Papert (1990) bricoleurs, mid-course adjustments were apparently an accepted aspect of his problem-solving style. Steve also volunteered that he had made copies of all the programs so that he could use them as models when he was coding in the future.

As Steve correctly completed the GetInput module, he duplicated the main module variable names in the formal parameter list. When asked why, he immediately changed the formal parameter names. He next constructed the CalculateWage procedure (making all three parameters variable), and continued with the Display procedure, offering that he was going to make the parameters VARs "just to be on the safe side." One could assume that Steve had discovered he could get programs to produce the correct results if he used variable parameters. In this case, though, he thought the Display parameters should be variable because "you need to get the information from the procedure." Steve understood that a variable parameter is required when the information is returned from the procedure; like other participants, he did not always know when it needed to be returned.

Although Steve's modularized version of the payroll program consistently used different actual and formal parameter names, the formal parameter names were identical in all procedures, just as they were (inappropriately) in the preceding task. Moreover, his BQ assignment had used identical names throughout. If Steve's hypothesized intuitive conception of variable parameters is accurate, then he found a way to reconcile his conception (that information is conveyed between procedures via a procedure-to-procedure connection established by identical formal parameter names) with Greene's requirement that actual and formal parameter names differ.

One remarkable trait Steve possesses is his perseverance. When he compiled the program the first time, he encountered the compiler error message *String constant exceeds line* because he had misplaced an apostrophe in a WriteLn. Steve thought the cause of the error to be a WriteLn that produced too long an output line. He admitted having longer statements in other programs, but conjectured that the rules might be different for WriteLns in procedures. In an attempt to rectify the problem, Steve changed the format specifiers in a different WriteLn than the one with the error. When that tactic did not help, he admitted being stumped. However, he elected not to stop. In answer to Steve's request to execute the program, I noted the program had not compiled, and proposed that it must have a syntax error somewhere. A short time later, after consulting the on-line help, Steve heaved a heavy sigh of relief and said, "I know. I know. I know." He added a second apostrophe to a WriteLn that should have had none. Although the program compiled, it displayed the incorrect string constant Steve had just constructed. This time he recognized his error, and without further ado, corrected the offending WriteLn. As Steve finished the task, the audio tape ran out. He had worked for 45 minutes on the modularization task.

Task 4

After receiving the instructions for the predict-the-output task, Steve talked himself through the process. "I'm looking at the procedure first, just to see what the procedure does," he said. "Basically all procedure Increment does is keep a count." He seemed to have provided his own context for the abstract task, attributing the familiar task of counting to the procedure Increment. He continued, "It's going to assign N a value of 3.

It runs through the procedure. A and B, and it's N and N. So it's 3. So it's basically 3 and 3." Steve's words demonstrated his understanding of the actual-formal parameter connection. He continued, "It replaces the A with the N or 3. It says 3 is equal to 3 plus 1, so 4 basically. And the same thing: 4." To Steve, who ascribed to Increment the function of counting, 4 was not only a reasonable answer given the communication perspective, it was the logical outcome for the procedure.

Offering that changes made to A and B were reflected in the main line because they were VAR parameters, Steve concluded by declaring that the answer would have been 3 if the procedure had value parameters. Both this and the previous analysis task (Task 1) contained only one procedure that was called once. For these tasks, Steve recognized without difficulty the effect of the value and variable parameters, given the communication perspective.

As the interview drew to a close, Steve once more explained the difference between value and variable parameters, saying, "A value parameter is passed only from the main line to the procedure and not back. Variable parameters pass back and forth between procedures and the main line, depending on how many procedures and whatnot." His phrase "depending on how many procedures and whatnot" is consistent with the hypothesized procedure-to-procedure link.

Asked if he had any mental image of why a parameter is changed one time (when it corresponds to a variable parameter) and another time it is not (when it corresponds to a value parameter), Steve alluded to the in-class simulation with the paper and the notepads:

> When it's a piece of paper, it's a value parameter. When it's a notepad, it's a variable parameter. The procedure is allowed to write one thing on a piece of paper, but not to send it back. If it receives something on a notepad, it could turn the page, write it on another piece of paper, and send it somewhere else. The next procedure would turn the next page and write something on there, but if there isn't a next page, then it can't send anything out.

Again, the phrase "send it somewhere else" is consistent with the interpretation suggesting procedure-to-procedure linkage. Steve concluded by saying, "You know, that's just the way I picture it. It helps a lot."

ReadLn

Without much hesitation, Steve identified the ReadLn parameters as variable "because with the ReadLn, once you ask for something and it's stored in the computer memory, it can be used throughout. In other words, once it's stored, it knows that it is supposed to be associated with A, for example." It almost seems that Steve was imagining that VARing a parameter made it global (known "throughout"), an accurate depiction if all procedures employ identical formal parameter names when referring to the same memory location (as Steve's programs did).

Steve first stated that if ReadLn's parameters were value parameters, the variable that is the actual parameter would contain the value that had been typed in. Then, however, he insisted, "I know it can't do the same thing as a VAR parameter. I know value and VAR parameters can't do the same thing. I know a value parameter can't pass information between procedures." Steve pondered some more, and although he did not clearly express what would be in the variable, he did decide it would not be what the user entered. The unseen ReadLn had given him a slight pause, but Steve was able to work through his thoughts and respond appropriately, perhaps because the correct answer to the question did not cause a conflict with his hypothesized procedure-to-procedure link.

Case Study: Moe

A Capsule of Moe

The most challenging participant to analyze in terms of his understanding was Moe. He could construct programs that produced the correct answers. However, as he believed that variable parameters were required to pass information between procedures, Moe tended to overuse variable parameters (his original programs used them exclusively). Moreover, he mistakenly believed that formal and actual parameter names could not be the same. His word choice concerning the value and variable parameters rendered the data difficult to interpret; it differed from the instruction and from that used by other participants. There seems to be some evidence that either he imagined a procedure-to-procedure link accomplished via variable parameters, or was confused regarding the effect of the value and variable parameters, or possibly both.

Moe was able, after some reflection and elaboration, to successfully solve the first analysis problem and to describe ReadLn's behavior. However, his solution (3) to the second analysis task again suggested a flawed conception of the parameter process. Moe is a formal thinker, and his general problem-solving approach on the interview tasks identified him as having an analytical style.

Moe concisely expressed his perception that he did not understand how parameters are passed, and seemed discomfited by the lack of understanding. Mayer's (1985) observation that students will construct their own mental model if they are not provided with one was confirmed by Moe's construction of a flawed model of the parameter process. Given the cognitive dissonance that Moe expressed and his erroneous self-constructed mental model, it is reasonable to hypothesize that he would have profited from exposure to a transparent mental model in a form he could internalize.

Background

A freshman who had no declared major or minor at the beginning of the CIS 110 semester, Moe is a male who is under 21. Offering that he had for some time been interested in computers, Moe said he enrolled in CIS 110 because it is the first programming course in the sequence. At the semester's outset, Moe indicated he hoped to succeed in the course, but had some concerns (He earned an A-). Moe is a formal thinker, scoring 23 on the IPDT. He missed only question 60, the somewhat ambiguous distance question.

Reactions to the Course

During the second interview, on November 14, Moe described the class as "really interesting" and sometimes a little too challenging, claiming the topics preceding the "more complicated procedures" had not posed undue difficulties. He named the BQ assignment as his favorite "because it took the most knowledge to do." His version, however, incorrectly passed every parameter as a variable parameter. Commenting that he realized his error when the assignment was returned (several days prior to the second interview), Moe would, however, replicate the pattern of using variable parameters exclusively in the upcoming modularization exercise. Moreover, both parameter laboratory exercises contained some incorrect value-variable choices. Moe missed four questions (of 50) on the mid-term examination that preceded the second interview by several days; all were parameter-related questions.

Parameters

Commenting on his difficulties with the parameter construct, Moe admitted that he had “a hard time understanding how the information was passed from one procedure to another and then back into the main code.” Moe’s expression, “passed from one procedure to another and then back into the main code” provide initial evidence of an imagined procedure-to-procedure conception of the parameter process. Claiming that he had “figured it out,” Moe said:

> I don’t necessarily know how it does it, but at least now I know how to make it do it. I would like to know how the information was passed from one to the other. I know that it is, and what you have to do to get it to do that, but I don’t know how it does it. Sometimes it bothers me a little.

When describing procedures, Moe distinguished between value parameters, variable parameters, and the output of a procedure. His description of parameters defined them as being composed of variables, and he used the phrases “order of the variables used in the parameter” and “variables are passed between parameters.” Hence, it appears Moe interpreted parameter to mean the entire list. “The output of the procedure,” Moe offered, “is what you need back or what the user ultimately wants.” Evidently, Moe referred to the value that is returned to the calling module as the “output of the procedure.” For example, he claimed, “The output is passed in the order of whatever the variables were in the procedure," and "the actual parameters will output in the same order that the formal parameters are."

Moe described a value parameter as "a number where the original number is not important." Describing changes to a value parameter, he also said, "When a value parameter is passed into a procedure itself, and it's changed in the procedure, the same number does not go back to the main line." He continued: "The original number that was passed into the procedure, the main line does not need it." Had Moe not distinguished between "output of the procedure" and value parameters, his statements would suggest he imagined value parameters destroying the value in the variable designated by the actual parameter.

Of variable parameters, Moe said, "The procedure can change it, but it will also send the original back, or at least the main line keeps the number somewhere. The original number is important and the output can be important, too." He also described a variable parameter as "assigning that number temporarily to a certain space in memory." Moe's descriptions of value and variable parameter suggest that he imagined variable parameters "remembered" and value parameters "not remembered," in exact opposition to the authentic processes. However, the interjection of "output" and the abuse of the terminology render the data difficult to interpret.

Analogy

During the analogy portion of the interview when Moe was first asked to apply his understanding of parameters, he offered:

> In situation one, the Mona Lisa that the instructor brought in is the variable parameter and the completed assignments that the students take home are the output of the procedure. In situation two, the original Mona Lisa is a value parameter, and the restored Mona Lisa is the output of the procedure.

Moe once again used the phrase "output of the procedure" to refer to information that is passed out of the procedure. Hence, in the first situation Moe seemed to equate the students' paintings with information being passed out or returned, perhaps relating to his phrase "what is important, what the user wants." In situation two, the restored Mona Lisa was returned. His analogy was consistent with his descriptions of the value and variable parameter terms. In accordance with his understanding, the original Mona Lisa in situation one was important and was unchanged after the procedure, and the original Mona Lisa in situation was not important and was not "remembered."

Task 1

Given the task of choosing the correct procedure heading line, Moe quickly established that the procedure exchanged the two values. He recognized the expected output if the procedure performed correctly. Then, observing that "Both outputs are 16," he just as quickly discerned that one parameter would have to be value and one would have to be a variable. Exuding a heavy sigh, Moe confessed, "Now all I have to do is figure out which is which. I've never figured out exactly how it works." He continued:

> I think that I [the participant] want I to be a variable and J to be a value so that when I is sent to the procedure, it will come back the same. When J is sent back to the procedure, it will have been put through the procedure and have been changed. J will have been changed to 16, and I--well it may have been changed to 5 in the procedure--will stay 16 in the main line.

Moe's first analysis attempt reflected precisely the descriptions he had provided earlier. Changes to a variable parameter are not reflected in the main module, while changes to the value parameter are. Continuing, Moe correctly associated the actual and formal parameters, demonstrating that he understood the connection. Tracing through the code a second time, he observed:

> I think I have it backwards. I think number one is the correct procedure. Unless I can sit down at a computer and test them a little bit, I couldn't tell you which one it is. One or four, it is either one or four.

Once more a student acknowledged relying on the computer for assistance. Moe had earlier admitted to not understanding the parameter process, but had "figured out how to make it work" (by using variable parameters exclusively) during the construction process. He had also apparently found he could rely on the electronic aid in the analysis tasks. As a result, Moe had not been forced to confront his misconception. At the interview's conclusion, Moe, given the opportunity to choose between procedure line one and four, amended his earlier statements. He correctly selected the first procedure line, offering an appropriate explanation for his choice:

> The variable N1 is a value parameter, which means it's not going to change anything in the mainline program. N2 is a variable [parameter] which would mean that it is going to change something in the procedure and output the 16.

Apparently Moe was either constructing or consolidating his understanding of the parameter construct as the interview progressed, thus perhaps profiting from the opportunity to elaborate. As he admitted to guessing on similar test questions the previous week, the interview may have been the first time Moe earnestly pondered his conception of the parameter process.

An examination of the problem-solving situation suggests that Moe adopted an analytical approach, attending first to the program's purpose. He either accepted, given the procedure name Swap, or discerned, given his lab experience, the procedure's task. Only later did he examine the details of the procedure body.

Task 2

The third interview, scheduled for November 22, commenced as Moe undertook the task of completing the procedure heading lines. Addressing the procedures in the logical order of their execution, Moe consistently chose correct formal parameter names, indicating he understood the need for identifier (name) consistency within a module. After correctly identifying the GetInput parameter, he added:

> I'm trying to decide whether it's a variable or a value parameter. I think it's a variable parameter because it's going to be used in this procedure and in the procedure to calculate the volume. It has to be passed from one to another without being changed.

The phrase "passed from one to another without being changed" again suggests that he imagineda direct connection. Tracing the program's execution, he correctly identified the changes to the mainline variables, Depth, Width, and Length.

Upon further examination, Moe decided that the CalculateVolume parameters Len, Wid, and Dep should be value parameters because they are "changed in the procedure and not used again." As the procedure body's only statement

```
Volume := Len * Wid * Dep
```

employs, but does not change Len, Wid, and Dep, the phrase "changed in the procedure" generates a question regarding his possible misunderstanding of the assignment statement.

Moe correctly identified Len, Wid, and Dep as value parameters; however, it may have been for the wrong reason. As he was examining the problem in the logical order of execution, his statement "not used again" was likely referring to the non-reporting of the rectangular solid's dimensions in the DisplayOutput procedure. A probe to determine if a version of DisplayOutput that included the dimensions caused Moe to change the parameter status of Len, Wid, and Dep would possibly have revealed further clues to his misconception. Referring to the procedure statement, Moe observed that "Volume is calculated here and is output to the main line." He correctly assigned it a variable parameter status because it "is used in the main line and in the procedure."

Moving to the DisplayOutput procedure, Moe decided that Volume needed to be a variable parameter because it was "passed between procedures in the main line." This time his reasoning resulted in the wrong choice. Moe's reasoning regarding the appropriate use of variable parameters could account for their overuse. It may be that he reasoned that information used in the program subsequent to the execution of the procedure in question needed to be communicated via variable parameters. As essentially every variable was ultimately reported in the output of the programs to which the class was exposed, the programs (according to Moe's rule) needed variable parameters exclusively.

Task 3

Next Moe completed the modularization exercise. After examining the program to discern its task, Moe combined the input and processing functions into one procedure. Having completed that, he constructed an output procedure. Moe's reasoning was consistent; both procedures used variable parameters exclusively because "they're passed between procedures, in the output." However, his phrase "Procedure into the main line, outputting HoursWorked, PayRate, and GrossPay" as he referred to his input-calculate procedure, diminished support for the hypothesized procedure-to-procedure link.

Moe erroneously believed that actual parameter names (identifiers) are required to differ from formal parameter names. (A review of his original constructions revealed that he strove to ensure that no identifier was duplicated.) Moving statements, he had neglected to change the identifiers in the moved procedure statements, while he carefully constructed formal parameter lists with formal parameter names that differed from the actual parameter names. As the identifiers used in the declarations did not match the use in the procedure statements, the program did not compile. Moe, however, attributed the failure to compile to the fact that the identifiers used in the procedure statements were identical to those in the main module statements. Moreover, he added, "If it does compile, you'll get the wrong input, output anyway, most probably." Moe's statement, which implies that identifiers (parameter names) govern the communication process, suggests a fundamental misunderstanding of the process by which information is communicated among a program's modules.

Task 4

Advancing to the second analysis task, Moe first sought an overall picture of the program's function. As he noted that N, which he observed was "assigned a value of 3," would "be put into A and B," Moe realized that the value of N was being communicated to both A and B. However, he continued:

> But the WriteLn (N) seems like it will just write 3 because it's only outputting one N. It doesn't make sense to me how the procedure can . . . because in the WriteLn there's only one N. According to the procedure, you're going to want two outputs, or two different . . . you can't have two values in the same variable name. So to me, it would make sense [that] either there's a runtime error or else N still equals 3. It just doesn't make sense to me otherwise.

One interpretation of Moe's remarks suggests that his original description of variable parameters, in which the original value of the main module variable is "remembered," still guided his thinking. Simply stated, there was no procedure to receive and display the changed value of the variable parameters A and B.

Conversely, Moe's analysis could be interpreted to suggest that he seemed to be thinking that A and B could not both change N because "you can't have two values in the same variable name." It may be that he was experiencing a conflict between what he knew about variables (that they can hold only one value at a time) and the communication view of the parameter process (that variable parameters return values to the calling module). His conception of variables prevailed, he abandoned the vulnerable parameter conception (one he had already admitted to not understanding), and he opted for the only possibilities that made sense to him: runtime error or 3.

ReadLn

The interview culminated with the questions related to the ReadLn procedure. Asked about the ReadLn, Moe first conjectured that it would be a value parameter "because you put [it] in the variable N, and that's the only place it's going to be used. You might use it in the main line." Moe's earlier statements that "variable parameters pass values between procedures" would account for his initial response. Then he altered his statement, observing that "A ReadLn can pass things to a WriteLn. Then it would make sense to me that both of them [the parameter lists of the ReadLn and WriteLn procedures] would be variable because they're going to pass and not change. Interpreted in terms of his earlier statements regarding variable parameters, Moe's account was reasonable (Values are passed "unchanged" between procedures via variable parameters). Elaborating further, Moe correctly decided that WriteLn could have a value parameter because "all the WriteLn does with it is output it to the screen or printer." He continued:

> But the ReadLn can pass it either into an equation or into a WriteLn or wherever, so the ReadLn would be a variable and the WriteLn would be a value.

In a few brief moments, Moe moved from believing the ReadLn would have value parameters to believing that both standard procedures needed variable parameters, to correctly recognizing that ReadLn required a variable parameter while value parameters would suffice for the WriteLn. His succinct answer precisely described the standard procedures' behaviors. One could hypothesize that the interview process, in which Moe was encouraged to verbalize his thoughts, provided the scaffolding Moe needed to progress toward principled understanding. Moe, however, was not asked to apply the knowledge the ReadLn behavior.

Case Study: Jason

A Capsule of Jason

Jason was able to construct correct programs as long as he was allowed to use the same formal and actual parameter names, and his nearly perfect first compilation attempt in the modularization task suggested he was not relying heavily on the computer to find his errors. During the construction tasks, Jason chose value and variable parameters correctly. However, his repeated comments that "value parameters are used when only that procedure needs it, and variable parameters are used when the value needs to be sent to other procedures" suggest that the choices were not consistently made for the correct reasons.

Jason's reliance on parameter names that duplicated the main module's variable names camouflaged an important misconception. He failed to choose appropriate parameter names during the hand-construction task; a possible interpretation is that he was not cognizant of the scope rules for parameter names. A more cogent argument asserts that Jason imagined a direct procedure-to-procedure connection in which variable parameters with identical names provided the link. In harmony with his conception, Jason appeared to have expunged the role of the actual parameter list in the parameter passing process. The evidence provided by the interviews and his class work supports that argument.

Jason, the formal thinker, consistently demonstrated an analytical problem-solving style during the interview tasks. If Turkle and Papert (1990) are correct, then Jason--

more than any other participant--should have been secure and comfortable with the abstract, rule-driven, communication representation of the parameter mechanism. Nevertheless, Jason constructed a faulty conception of the process. In view of the image that he appears to have constructed, it seems likely that he would have profited enormously had he adopted the shared memory-location viewpoint of the parameter concept. Thus, the case of Jason lends strong support for Mayer (1985) and Du Boulay's (1986) position regarding the value of an authentic mental model.

Background

Jason, a sophomore under the age of 21, is a male majoring in paper science, a discipline that requires CIS 110 as a service course. At the CIS 110 course's outset, Jason indicated that he did not plan to elect the next computing course, and he did not. The only participant who judged the computing field to have little or no career potential for him, Jason indicated his confidence in his ultimate success in the programming course. The A- he earned in the course was higher than his overall B- average.

Indicating that he had worked extensively with computer applications and had used a computer regularly during a recent paper mill internship, Jason recalled copying "tiny" BASIC programs that came with an early computer. He also reported having positive preconceptions regarding the CIS 110 course. Friends who had taken the course previously described it as not difficult, only time consuming.

Beginning some time in the second week of November, just as the parameter instruction was concluding, Jason missed approximately one week of school because of emergency surgery. He did say, however, that he did not miss any of the actual classroom instruction on the parameter concept. (The class was canceled on November 10, the test was administered on November 11, and Jason was apparently back for class on November 15.) It is difficult to speculate how much impact his weakened condition and extended absence at this crucial time during the semester may have had on his class work and his participation in the study. Jason is a formal thinker, scoring 22 on the IPDT. He missed questions 54 (a classes question) and 59 (a distance question).

Reactions to the Course

Agreeing to a second interview on November 16, the day after he returned to school, Jason described the course thus far as interesting and time consuming, but "not too tough" and "not real boring for me." He identified what he liked best about the course:

> You have to do everything logically in order and steps and everything. Kind of everything you do in a program makes sense. Everything has its own little place and step, and has its own job to do in a program. It's kind of neat.

His observation about the best-liked facet of the course provides a clue to his learning style. Moreover, when asked which instructional strategy had proved most helpful, Jason (in contrast to other participants who chose concrete examples such as the laboratory exercises or the simulation) alluded to Dr. Greene's summation.

Jason indicated an earlier assignment (one that had focused on the selection control structure) as his favorite, correctly observing that it was the first complicated individual assignment. Acknowledging that he felt rewarded when he was able to complete the assignment successfully, he also commented that simple procedures were "pretty easy." The BQ assignment, however, posed some frustrations for him, and he described it as his least favorite of the assignments. Asked to describe his frustrations, he offered these observations:

> I got to the end, and I figured I would have a pretty decent start on it. When I went to run it, it didn't work right. The calculations were wrong, for one thing. I know that. It was kind of odd. I figured that would, that should be one of the things that would come out for sure, and then it didn't go into the loop right or something like that. It just didn't work right.

Despite Jason's reported struggles, the parameter passing in the BQ assignment was correct, except that it used identical formal and actual parameter names. His Flipper program in the Lab 7 exercise was also incorrectly done, while the parameter passing in the other Lab 7 tasks was correctly done. On the test (which he took late), he missed three parameter-related questions: 10, 12, and 30. Two questions involved a procedure with value parameters; one involved the actual-formal parameter connection. Jason's pattern of errors, when combined with the performance in the interview tasks, will prove significant. As he apparently did not turn in the second parameter laboratory exercise (Lab 9), it may be pertinent to note that the only correctly done programs that required different actual and formal parameter names were the two exercises in Lab 7.

Parameters

When asked to describe the topic that challenged him the most, Jason indicated the parameter topic. "That was tougher," he admitted, and went on to confess his confusion concerning which of the parameters had to get passed on to different procedures, and which ones "had to be called at the end in the main program for the procedure heading, what parameters had to be put in there." Later he defined his difficulties in greater detail, claiming:

> It's not so bad knowing the stuff that has to go to other procedures. I get mixed up on the actual parameters, which ones you have to call on at the end when you call it up, and I can't remember. It's the formal parameters that go right after the procedure headings. That's where I get mixed up mostly.

It seemed at first that Jason's difficulty might be the terminology. However, his statement that "value parameters are used when only that procedure needs it, and variable parameters are used when it the value needs to be sent to other procedures" suggested he had a fair grasp of those terms. Expressing belief in his ability to recognize the difference between the value and variable parameters, Jason, however, confessed:

> I'm not sure in the value parameter, if that comes from a different procedure or if that's one that's made in the procedure itself, if they come in from a different procedure or if they are made in that procedure.

When asked what is provided in the actual parameter list, Jason replied, "That's what I'm not sure about. I really don't know that." Responding to a question about variable parameters, Jason said, "Those parameters are able to be passed on to other procedures. They can be moved to other procedures." Jason's words described his conception: variable parameters are "passed to other procedures."

When queried for a definition, Jason incorrectly described a parameter as "a variable within a procedure." As he had expressed confusion concerning where the value parameters are "made," his response at first suggested that Jason lacked virtually any understanding of the parameter concept. However, his elaborated answer: "The parameter is known in other procedures, and it can be known in the main" dispelled that notion. Jason understood that parameters are a means of sharing information among program modules. While the reference to "can be known in the main" may only have been an unfortunate word choice, the bulk of evidence suggests otherwise. The response was congruent with his repeated description of variable parameters as being passed between procedures. As Jason several times repeated his statement about not understanding "where value parameters are made," it is more likely that he did not understand the role of the actual parameter list in the parameter communication process. His problem-solving activity in the upcoming tasks lent support to this interpretation.

Analogy

Posed with the Mona Lisa analogy, Jason correctly identified the painting in situation two as similar to a variable parameter, saying:

> If you have one value and something has to be added to it or subtracted or you need some kind of new value from the original, it has to be sent to a different procedure and then pulled back out.

Once more, Jason referred to variable parameters as being passed between procedures, providing more support for a direct-link theory.

Describing the Mona Lisa in situation one as a global variable, Jason offered, "All other procedures have access to it, and it stays unchanged." With surprising precision, Jason described the problem with a global variable (one to which all procedures have access). Therefore, it became important to discern whether he connected the parameter role in protecting global variables from inadvertent alteration with the need to prevent the students (who have no reason to apply their paintbrushes to the original Mona Lisa) from doing so. It is possible he made the connection. However, in view of other evidence, it is more likely that Jason--who did not know "where value parameters were made," and who seemed incognizant of the actual parameter list's role in the communication process--believed the value parameters to be "made" from a global variable.

Task 1

After receiving instructions for the first analysis task (selecting the correct procedure heading line), Jason looked first to the main module of the program. After remarking, "I see there's a 5 down there, and there's no 5 in the output," Jason turned to the procedure body, and described an algorithm that would equate the values N1, N2, and T. Asserting that he did not know which of the procedure heading lines would be appropriate, Jason claimed he had no ideas, and asked to stop.

Initially, Jason's request to discontinue the interview seemed to reflect the "stoppers" Perkins et al. described in their 1986 study. "Lacking a ready answer to the difficulty, the student not only feels at a complete loss, but is unwilling to explore the problem further" (p. 42), they contend, but this explanation seems unlikely in Jason's case. Given his success in the demanding paper science curriculum, a slightly more plausible interpretation would be that he was exhausted or rushed or perhaps too polite to ask to be excused from participation in the study. Another equally probable explanation proposes that Jason, who had judged a computing career to have little or no potential for him, and who had no plans to continue in the discipline, attached no importance to the abstract task.

A more interesting interpretation is one that assumes that Jason was engaged, that he knew what values were passed to the procedure through the formal parameter list, and that he knew what happened to the variable parameters in the list, but that he did not understand the workings of the procedure. Referring to the task, Jason acknowledged that he had seen something similar in the laboratory exercise, but claimed he did not remember the procedure's specifics. A review of Jason's Flipper program disclosed that the instructions there were incorrectly coded, resulting in no swap. Professor Greene, although he had not deducted points, had written correct code beside Jason's erroneous code. Perhaps because points had not been deducted, Jason had not recognized the significance of his error or tried to remedy his understanding. More likely, Jason--on his second day back in school--had not yet looked at the graded exercise. For whatever reason, it seems possible that Jason was unaware of the errors in his Flipper program as he undertook the interview task.

Jason would have needed to trace the swap algorithm correctly for the problem to make sense. If he traced the algorithm such that N1, N2, and T all ended up equaling 16, then three of the four procedure heading lines would have provided the required output. If, on the other hand, he traced the algorithm such that N1, N2, and T ended up equaling 5, none of the procedures heading lines would produce the provided answer. An interpretation that asserts that Jason understood the parameter construct, but misunderstood the effect of the procedure's statements, could account for his confusion on the first interview task.

Combined with other evidence, another explanation seems equally likely, an explanation that proposes Jason knew neither the values that were passed to the procedure through the value parameters in the formal parameter list, nor the destination of the variable parameters in that list. Contrary to most of the examples he had seen previously, there was no input procedure to obtain the values and pass them to the swap procedure. Furthermore, there was no output procedure to which the swap procedure could pass its information to be displayed. If Jason did not understand the role of the actual parameter list, he would not have known where to begin, an interpretation of the situation that seems apt.

What is the most persuasive explanation for Jason's behavior? Given the preponderance of the data, it is likely that Jason could not trace the code, and that he neither knew what was passed to the procedure in the formal parameter list as value parameters, nor did he know what happened to any variable parameter information.

Task 2

The November 29 interview commenced when Jason began the procedure heading line-construction task. He first looked at the main module, then turned to the first procedure on the page. In his construction (which follows)

```
PROCEDURE CalculateVolume (     Len : Real;
                                Wid : Real;
                                Dep : Real;
                          VAR Volume: Real);
```

of the procedure heading line for CalculateVolume, Jason correctly named Len, Wid, and Dep (because of their use in the assignment statement) and made them value parameters because, in his words, "they are only going to be used in this procedure." Volume he made a variable parameter, citing as his reason, "it has to be sent, available for the DisplayOutput procedure to use." In this instance, Jason constructed the procedure heading line correctly. However, it seems possible that it was a coincidence due to the fact that DisplayOutput reported only the volume of the rectangular solid and not the dimensions. Jason's repeated assertion regarding the need for a variable parameter--when the information must be available to a subsequent procedure--seems to imply that if DisplayOutput had reported the dimensions, he would have made those parameters variable also. Unfortunately the problem was not posed.

Next, Jason worked on DisplayOutput (which follows), the second procedure on the page.

```
PROCEDURE DisplayOutput(     Volume: Real);
```

He gave the procedure a value parameter because "it didn't need to be sent anywhere." However, he named it Volume rather than Answer, the identifier used in the procedure statement, because, in his words:

> It has to come from the above procedure [CalculateVolume]. Volume was used as a variable parameter up there [Volume in CalculateVolume], so it has to be sent down to this procedure DisplayOutput.

Jason's observations implied that he imagined a direct variable-to-value parameter connection between procedures, a connection that was established with identical parameter names. (Jason's hypothesized conception as somewhat resembled the actual process that occurs when nested procedure calls are employed in a program.)

Last, Jason examined the GetInput procedure, the third procedure on the page. The main module procedure calls and Jason's procedure heading line follow.

```
GetInput (Depth);
GetInput (Width);
GetInput (Length);

PROCEDURE GetInput (VAR Len :Real;
                    VAR Wid : Real;
                    VAR Dep : Real;)
```

He incorrectly gave the procedure three parameters--Len, Wid, and Dep--which he correctly identified as variable parameters "so they can be sent to the procedure to perform the calculation," the CalculateVolume procedure. A general question about the appearance of the GetInput parameter list did not provoke further attention to the number of parameters in the list.

One interpretation of Jason's parameter name choices in the GetInput and DisplayOutput procedures suggests that he was not sensitive to the fact that the scope (the parts of the program that may use a particular identifier) of a parameter name is limited to the procedure in which the formal declaration appears. It also suggests that he may not have realized the formal parameter list provides the declarations for the variables used in the statements of the procedure. Jason used identical actual and formal parameter names (meaning that every parameter name matched a main module variable name) in every task that allowed him to do so. Moreover, he commented that he always used the same names. Doing so may have camouflaged the fact that Jason did not understand the scope of the identifiers. If there were no other evidence, this would be a reasonable and sufficient explanation for Jason's choices.

Conversely, Jason may have understood the scope rules. If he understood the scope rules but did harbor the hypothesized misconception regarding the direct procedure-to-procedure, name-connected link, his conception of the parameter process would have clashed with his knowledge of the scope rules *if* the various procedures used different formal parameter names to refer to the same memory location. His consistent use of identical variable and parameter names throughout a program may have allowed him to reconcile his naive parameter conception with the scope rules. Moreover, the tactic would have allowed him to construct programs that compile, execute, and produce the correct answer.

Interpreted from the perspective of Jason's hypothesized name-controlled, direct-connection link, his parameter choices were appropriate. CalculateVolume provided a Volume to DisplayOutput; thus DisplayOutput needed a Volume to receive it. CalculateVolume needed Len, Wid, and Dep; thus GetInput needed to supply them. It is noteworthy that Jason examined the procedures in the order of their physical appearance on the page, rather than the logical order of their execution. Apparently, that order determined the name choices.

Task 3

Jason began the modularization task by constructing a GetInput procedure, to which he moved the input prompts. Next he turned his attention to the CalculatePay procedure, making HoursWorked and PayRate value parameters because "they won't have to go any further than this procedure." GrossPay needed to be a variable parameter because "it has to be used in another procedure that's going to display it." Jason's words provided a second chance to inquire about the need for variable parameters. The question: "If you were accumulating totals of HoursWorked for multiple employees somewhere (i.e., using HoursWorked in another procedure), would HoursWorked need to be a VAR parameter?" evoked the (correct) response: "No, I don't think so because it is a VAR up here [in the GetInput procedure]." His statement seemed to indicate that he understood when a variable parameter was needed; he just did not know what happened to it. That interpretation is consistent with his correct value-variable choices in the BQ assignment.

Jason next examined the display procedure, and concluded with an instructions module. He checked over his work before compiling; he was also the only participant who left the original program essentially intact and available for review (by enclosing the unnecessary Pascal statements in comments that become transparent to the compiler during the compilation process). Jason's first attempt contained only one small syntax error, an overlooked data type in a procedure heading line. As previously noted, however, he uniformly used the mainline variable names as formal parameters.

Task 4

After looking briefly at the program in the code tracing exercise, Jason summarized his intended approach:

> First I'm going to see what the procedure does. Now I'm looking at the main line. Both A and B are going to be stored as N, which is 3, so WriteLn

> (N) is going to be equal to 4. A and B are both going to be equal because they're both stored as the same value, which is given right here as 3 in the main line. The procedure adds 1 to N, which is 3.

Jason recognized that the changes to the variable parameters in the procedure Increment would be reflected in the calling module, the main module. A diagram he drew, shown in Figure 2, illustrates that he understood the correspondence between the formal variable parameters A and B and the main module variable N.

A B
↓ ↓
N N
= 4

Figure 2. Depiction of Variable Parameter Association

In the absence of other evidence, Jason's response could mean that he correctly understood the communication aspect of the variable parameter concept. That both A and B returned a value of 4 might have allowed Jason to reconcile his perception that both were "stored at the same value [location]."

In view of the accumulation of evidence, another interpretation of Jason's response is more cogent. The "global" variable N provided the value 3 for A and B. After the procedure added 1 to both, A and B replaced N. Jason was able to reconcile the different names--N versus A and B--because the main module only had one variable; it was the only choice.

Later Jason offered that a change to A or B changed N because A and B were variable parameters. Asked if he had any mental image of how that happened, Jason replied, "I don't know if I have an image. If I see a VAR, then I know it can be sent to other procedures." I [the interviewer] slipped and said, "Do you understand why, when you change A that it's really N being changed?" Referring to the diagram (Figure 3), Jason responded, "Yes. Basically I see these two numbers; N and N are in this order. That means A and B will be stored in that order. I just see two little arrows that go right there." Jason knew a correspondence took place between the main module N and the formal variable parameters, and he knew that changes to the formal parameters were communicated back to the calling module. However, he seemed to be imagining copies replacing the main module variable N, with the order determined by the position in the parameter list. The unintended intervention provided by the statement "A change to A is really a change to N" did not help Jason, as it contradicted the mental image he had constructed of the way variable parameters are passed. As a novice, Jason appeared to have resisted instruction contrary to his intuitive conception, a practice also noted by Linn and Songer (1991) in their study of science students.

As the unintended intervention had not provoked the feared disaster, it seemed reasonable to pursue the topic. To the question: "If they were value parameters, what would happen to changes made to A and B; that is, if the VAR were not there, what would N be?" Jason replied, "It would still be 4, I think. The VAR really doesn't . . . I don't think you need it in this procedure." Asked again: "If this were a value parameter, if A and B were value parameters, what would N be when this WriteLn (N) is executed?" Jason again answered, " It would still be 4." The question: "If it would still be 4, how do value and VAR parameters differ?" evoked the following response:

> If you have more than one procedure, say three, four procedures . . . if A and B would have to be sent from this procedure to a different procedure, then you need VAR, but this program is not big enough to need a VAR.

Jason's response was congruous with his other statements. Interpreted from the hypothesized name-controlled, direct procedure-to-procedure link, the parameters in the first procedure to be executed (the Increment parameters A and B) obtained their values from the "global" mainline variable N. As the program had no other procedures to which the value needed to be sent, variable parameters were unnecessary.

ReadLn

Asked whether ReadLn's parameters were value or variable, Jason responded, "VARs, because ReadLn has to be sent from somewhere else in order to get it." As the response did little to assist in understanding his thoughts, Jason was asked to apply his understanding of the ReadLn parameter list. Framed in terms of a modification to the previous analysis task (task 4), the problem was posed as follows:

> What if we changed this program a bit? Instead of the call to Increment, the main module assigned a value of 3 to N using an assignment statement, and then had ReadLn (N). Assume the user typed in a 6 and that the ReadLn's parameter was a value parameter. What would be in the variable N after the ReadLn? What would this WriteLn display?

Consistent with his response concerning the Increment procedure in the preceding task, Jason responded, "6." In Jason's mind the modified ABC program--without other procedures--was not, in his words, “big enough” to need variable parameters. The powerful image of Jason’s hypothesized procedure-to-procedure connection inhibited him from seeing the incongruity of his answer.

CHAPTER V

DISCUSSION

This chapter examines the data from the perspective of the original research questions, addressing the questions in order. It is followed by a summary of the major findings of the study and a statement regarding the significance of the findings. A discussion of the pedagogical implications of the findings, sometimes accompanied by recommendations for instruction, appears next. The chapter concludes by calling attention to the limitations of the study.

Research Questions

Action Knowledge

What action knowledge does the novice programmer bring to the study of parameter passing? It appears that a few pieces of action knowledge had an extraordinary impact, often negative, on the novices learning the parameter construct. This was most seen in terms that have computing meanings that differ vastly from the natural-language meaning of the words. Linn and Songer (1991) propose that the use of natural-world terms in scientific jargon inhibits the construction of intuitive conceptions. Others also have noted the problems associated with everyday meanings for computer terminology (Pea et al., 1987). Hence it is reasonable to conjecture that the action knowledge the participants brought to the learning situation may have interfered in the learning process.

Because some evidence suggested that not all participants had espoused the technical meaning of the word, “variable” is discussed first and at length. Terms specific to the parameter concept are addressed only briefly, as it was not apparent that prior knowledge of those terms posed serious problems for the participants. The information processing load, caused by the surfeit of terminology associated with the construct, appear to be of more concern.

Variable. Understanding the variable construct is fundamental to understanding the parameter construct. The Leestma and Nyhoff (1993) glossary defines a variable as "an identifier associated with a particular memory location. The value stored at this location is the value of the variable, and this value may be changed during program execution" (p. 967). As the variable construct is covered early in the CIS 110 course, it seemed reasonable to assume that students would bring a principled understanding of variables to the parameter learning situation. As Table 2 reveals, the expectation was only partially borne out.

Table 2

Descriptions of variable

Name	Description
Carol	A number that can change; a value that can be changed
Carl	Something that holds a value in memory; something that is relative to change; in the main line
Amy	Something that represents something else
Sammy	Something that stands for something else; you can assign a value, like to a string.
Sondra	A place in memory, and you are going to be able to store a value there; a place where you can store stuff.
Steve	Something found in the VAR
Moe	Something that can be changed to suit your needs; a user-defined value
Jason	A word that can be used to denote another word or some kind of number; a value that is used in the program

Although most participants did include in their definitions some phrase concerning change, only two (Carl and Sondra) referred to a variable as a memory location (see Table 2). It is likely that the everyday meaning of the term "variable" interfered with some students' adoption of the data storage notion inherent in the technical meaning. If the conjecture is true, then some participants may have been prohibited from developing a principled understanding of the parameter construct because they lacked the prerequisite knowledge.

Just as Carl and Sondra were the only two who included the notion of data storage in their variable definitions, they are also the only two who appeared to be even minimally aware of the shared memory-location perspective of the parameter construct. It may be that their conception of a variable sensitized them to the facets of the instruction emphasizing that view.

Beyond its specific application to the parameter construct, the conjecture regarding students' understanding of variables suggests that instructors cannot assume a student fully understands a concept solely on the basis of prior instruction, or demonstration of some facility using the concept. Referring to youngsters writing recursive Logo programs, McIntyre (1991) observed:

> Proponents of Logo point to the availability of recursion, reporting examples of young students using recursion in solving problems. . . . The ability of students to use recursion to draw complex and attractive patterns and designs is one thing; the ability of students to understand recursion as an elegant mathematical construct and a useful programming concept is another. (p.74) [underline in original]

Applied to the participants in this study and their use of the variable concept, the observation seems apt.

Professor Collier, the instructor of the one-credit Pascal class, provided a vivid account of her variable instruction for this study; the description is provided in the CHAPTER III section titled “Pilot testing of the interview protocols.” Given this elaborate definition, it would be reasonable to expect that the definitions given by students who completed that course (Amy, Sammy, and Steve) would reflect the notion of data storage regardless of whether the CIS 110 instructor had provided that kind of instruction. A perusal of those students’ responses (see Table 2), however, reveals little that resembles Collier’s statements. The data could be interpreted to mean that the instruction was not powerful enough to overcome the conception provided by the everyday meaning of the term “variable.” Irrespective of the skill of the teller or the quality of the tale, the data could also be interpreted as providing support for the constructivist position regarding the relative ineffectiveness of “telling.”

As stated previously, understanding the variable concept is essential to understanding the parameter construct. Hence, insuring that novices construct an understanding of the term “variable” compatible with its technical meaning is crucial. Current constructivist thought proposes that learners fashion meaning by talking, writing, and connecting ideas to their lives. Hence, variable-related programming instruction must structure opportunities for novices to do this. For example, programming students might develop variable analogies, draw pictures, or perhaps explain the technical meaning of “variable” to a non-programmer, and then reflect on the interchange. Another possibility is a semester-long, collaborative project during which teams design variable surveys, choose subjects, collect data, organize and present their results, and design and write programs related to and using the data they collect.

Terms specific to the parameter construct. As with variable, the natural-world meanings of the terms "procedure" and "parameter" differ greatly from the meanings employed by the computing community. As lack of parameter experience was the first criterion for participant selection, the students' conceptions of the parameter-related terms quite reasonably reflected the familiar meaning. The participants typically responded to the term "procedure" with variations of "a way of doing things, certain steps to take," or "how you do something." "Parameter" drew such responses as "lower bound and upper bound," "guidelines," "a form that something takes," "evolves around this kind of context," and "outline."

The terms "value parameter" and "variable parameter" seemed even less familiar to the students. Before the instruction, only two participants ventured guesses for the expression "value parameter." Carl offered "guidelines for a certain variable," and Sondra described it as a "parameter that evolves around some values." Carol and Sammy connected the term "variable parameter" with "range," while Sondra aptly described it as a "parameter that includes variables." Interestingly, Steve thought it to be a "parameter that changes." It appears that the everyday meaning of the variable portion of the term guided his thinking.

Summary. As classroom instructors cannot unilaterally alter the discipline's jargon, they must seek methods for overcoming the interference generated by the intrusion of everyday meanings into technical terms. One possibility is a parenthetic redefinition of the offending words with technical terms the students are less likely to know. The term "subroutine," for example, could appropriately serve as a re-definition for the term "procedure," and "data store" could substitute for "variable." While it might reduce the interference, the strategy would, however, likely exacerbate the serious problem caused by the surfeit of terminology associated with the parameter construct.

One conclusion is certain. Instructors, who have long been immersed in the computing community with its extensive jargon, cannot assume that novices attach the same meanings that professionals do to the technical terms pervading the instruction.

Intuitive Conceptions

What intuitive conceptions does the novice programmer possess regarding the parameter passing construct?

Naive conceptions. Du Boulay (1986) observed that learners, if not provided with conceptual models, construct their own mental models, models that may be impoverished. When applied to the parameter models the participants constructed, the perception seems appropriate. In accord with Linn and Songer (1991), who maintained that intuitive (flawed) ideas are largely derived from natural-world experiences, the intuitive conceptions of variable parameters developed by the participants in this study did seem to derive from natural-world occurrences. Carl, for example, envisioned a chalkboard, and Sammy imagined a hand grabbing the value and bringing it back (to the calling module). Although the words differed, all participants seemed to envision some notion of the value in the actual parameter being replaced by the value in the formal variable parameter at the procedure's conclusion. The obvious extension is that they incorrectly imagined the procedure receiving a copy of the actual parameter at the procedure's inception. Amy, for example, used the words, "N is copied into two different spots in memory" as she introspected during the second analysis task. From this evidence, one conclusion is certain. The opaque, one-way versus two-way communication model that permeates much introductory Pascal instruction apparently allowed none of the participants to construct a mental model that reflected the authentic variable parameter process.

If intuitive conception is interpreted to include fragile knowledge, then Sammy's and Sondra's repeated statements of "I'm not sure if . . ." provided evidence of their intuitive conceptions. The several allusions to difficulties with the terminology added more evidence regarding the fragile nature of some participants' understanding. Moreover, most participants erred when choosing between value and variable parameters either in their class work or during the interview tasks.

Syntax-related misconception. Sondra's efforts with the first analysis task divulged a misconception regarding the parameter syntax. He incorrectly believed the parameter lists `(VAR N1, N2: Integer)` and `(VAR N1: Integer; N2: Integer)` to have the same effect. As there is no reference in the research literature to the various forms the formal parameter list can assume (as related to learning), the topic is an ideal candidate for an experimental study. In contrast to fundamental misconceptions about the communication process, remedying syntax-related problems would not seem to pose undue pedagogical difficulties once they are identified. One instructional possibility is an exercise combining the same formal parameters into various lists. Learners could predict the behavior, execute the program to test their predictions, and answer questions about the behavior. Not only might such an exercise reveal or remedy syntax misconceptions, it could provide an opportunity for students to incorporate the parameter terminology in their writing and speaking.

Fundamental misconceptions. Steve, Moe, and Jason exhibited problem-solving behaviors that could be attributed to a fundamental misconception of the parameter process. Each made errors that could be interpreted as systematic applications of a parameter process that involved a direct procedure-to-procedure communication. The three participants appeared to imagine different processes; the common factor was the direct procedure-to-procedure link.

This finding has several important implications. First, Moe and Jason are both formal thinkers, and both consistently exhibited problem-solving behaviors that would be described as analytic. Steve is a concrete thinker whose problem-solving style appeared to be contextual. Thus it is not possible to conclude, as might have been expected, that concrete, contextual thinkers are more prone to misconception than formal, analytical thinkers. Tobias (1990) advised against assuming "too narrow a vision of what kinds of attributes, behaviors, and lifestyles the 'true' scientist displays" (p. 14). The evidence from this study suggests that her counsel applies to assumptions about novice programmers as well.

It is also important to note that, during the interviews, all three participants who seemed to foster significant misconceptions verbalized their inability to understand. Although the words varied, the common message was: "I do not understand." This fact suggests that students can provide helpful information about their understandings and misunderstandings if they are provided an opportunity. In recent years, student journals and other classroom assessment techniques have gained great popularity; incorporating their use in an introductory programming class could provide novices with the opportunity to say, "I do not understand."

Teachers might also be alerted to student difficulties by using a face-to-face grading scheme (Cooper, 1985). Obviously, an instructor teaching multiple classes of 30 to 35 programming students cannot engage in face-to-face grading with every student. However, journals or other classroom assessment techniques could identify students who would profit most from the interchange. The practice seems to offer many potential benefits. Students who explain a program's function, and justify design decisions, could obtain the same benefit that interview participants did: the opportunity to elaborate their conceptions. Moreover, one could hypothesize that students would be less inclined to submit a program they did not fully understand if they knew they might be required to explain the program's functioning to their instructor. Finally, the scheme ensures that students receive immediate feedback on their work.

Summary. If the replacement notion of a variable parameter is viewed as an intuitive conception, then it was one shared by all eight participants. Since that conception could reasonably be attributed to the instruction, further discussion will be postponed until the section on principled understanding. Beyond that, only two (Carol and Carl) of the eight participants did not display some form of intuitive conception or fragile understanding during the interview tasks or in the BQ assignment. One participant revealed a syntax-related misconception, and three displayed behaviors that could be interpreted as applications of fundamental misunderstandings of the parameter process. This accumulation of evidence suggests that the community of computer programming instructors may wish to examine its beliefs about the way the parameter construct has traditionally been taught.

Principled Understanding

How does the novice express principled understanding of the parameter passing construct?

Two perspectives. Basing their statements on their study of adolescents learning science, Linn and Songer (1991) noted that some scientific principles are inaccessible to students because the principles integrate a wide range of information, or because they are based on hidden mechanisms. It may be that the shared memory-location perspective of the parameter passing mechanism is such a principle. Linn and Songer advocated using "pragmatic principles" to assist students in the knowledge construction process. As Pascal textbook coverage, even coverage including the shared memory-location view, almost inevitably includes the one-way versus the two-way communication approach, some textbook authors may consider that the communication view is a necessary pragmatic principle for novices learning the parameter construct. Hence, for purposes of discussion, the copy versus the shared memory-location and the one-way versus two-way communication perspectives will both be considered principled.

As the instruction emphasized the one-way versus two-way communication outlook, it is not surprising that the participants adopted that viewpoint. By the third interview, all participants repeated the words of the text and the classroom instruction regarding value parameters. Although not unanimously successful, they also echoed the language of variable parameters, analogized appropriately, constructed modular programs, and analyzed problems for which that parameter model sufficed.

The students were less successful demonstrating principled understanding of the shared memory-location perspective. Understanding appeared limited to occasional employment of the terms "copy" and "original." No one, however, used the terms consistently, and no one was able to apply that knowledge in the second analysis task (Task 4). As novices rarely analyze the program correctly, and advanced programming students seldom err, it is reasonable to question why. One interpretation proposes that the code may be too "tricky." The brevity of the exercise may lull less sophisticated programmers into believing it trivial. Alternatively, it may be that the shared memory-location principle (upon which the task depends) is beyond the zone of proximal development of novices during the initial stages of parameter instruction. Still another explanation proposes that novices would be more successful with the task if they were provided with a more authentic model of the parameter process from the outset. Additional research may provide an answer to this question.

McGrath (1990) maintained that a good mental model must be useful to experts as well as novices. The communication parameter model allowed no one to solve the analysis task (Task 4) correctly, and allowed all participants to develop an inaccurate image of the variable parameter process. Hence, it is possible to conclude that the communication model alone is not sufficient (and perhaps not even useful), and that a better model would integrate the communication and the shared memory-location models of the parameter process. Two suggestions for doing so--within the structure of an introductory Pascal programming class such as the one studied--follow.

Cognitive benchmark. Linn and Songer (1991) reported that sometimes even pragmatic principles are incomprehensible, that novices need effective representations to help them understand abstract ideas. With few exceptions, the participants in this study referred to the classroom payroll simulation when recalling the instruction that had proved most helpful in providing an image of the parameter process. Given the apparent power of the simulation, one could theorize that a concrete representation of the shared memory-location parameter model, embedded in a familiar context, would help propel students toward principled understanding of that perspective (for the description of a simulation that concretely represents the shared memory-location perspective, see Appendix S). Bruner maintained that students need multiple representations that move from the concrete to the iconic to the symbolic (Marchionini, 1985). Accordingly, a revised version of the Leestma and Nyhoff (1993) example provides a graphical representation of the simulated payroll program, accompanied by verbal explanations (see Appendix T). The program itself provides the link between the concept and the Pascal syntax (see Appendix U).

Linn and Songer (1991) wrote that benchmarks are used as anchors or bridges to fill the gaps between novice and expert views. In their work with science students, a key experiment--where everyone understood the situation and the explanation--served as a benchmark, and was used to help students integrate action knowledge and intuitive conceptions. The multiple representations of the payroll program could serve as a benchmark to students learning about Pascal parameters.

Explicating the parameter process. Carefully structured to incorporate the guidance provided by the literature, the second analysis task (see Task 4, Appendix P) seems an ideal vehicle for forcing students to adjust their incomplete and sometimes inaccurate mental models of the parameter construct. Linn et al. (1987) found, for example, explicit instruction to be effective in their studies of AP classes. Moreover, Clancy and Linn's (1992) case study approach is built on the premise that students can benefit from expert knowledge if the expert thinking is made explicit (transparent). Van

Merrienboer (1987, 1990, 1992) and associates extolled the benefits of novices reading completed programs.

Accordingly, a version of the interview task (modified to incorporate the counsel of the literature) is found in Appendix V. Using the debugging tools provided in the Turbo Pascal Integrated Development Environment, students observe the action on the program variables as the execution progresses. The computer acts as the more able peer and makes explicit the parameter passing process as it happens, and the accompanying questions encourage mindful abstraction.

Problem-solving styles. In addition to the two parameter perspectives, the study explored another dimension of principled understanding, that of problem-solving styles. The participants demonstrated diverse approaches to problem situations, styles generally consistent with those described in the literature (Cheny, 1980; Johnson & Johnson, 1991; Turkle & Papert, 1990). Four students (Carol, Carl, Moe, and Jason) consistently displayed behaviors that could be described as analytical. Those students, for example, generally

- scrutinized the abstract analysis tasks without imbedding the code in a familiar context,
- did not need to stay in close contact with the details of a problem situation,
- discerned the general purpose of a program before examining the details,
- attended to the logical representation of programs,
- showed little evidence of relying on trial-and-error as a problem-solving approach, and
- alluded to control when describing their human-machine interactions.

Conversely, three participants (Sammy, Sondra, and Steve) typically displayed behaviors that were consistent with the literature descriptions (Cheny, 1980; Johnson & Johnson, 1991; Turkle & Papert, 1990) of heuristic or concrete (contextual) learners. Those students generally

- placed the abstract analysis tasks in context before analyzing them,
- attempted to remain in contact with a problem's details,
- often examined the details of a problem at length before discovering its purpose,
- occasionally seemed bound by a program's physical representation,
- regularly employed trial-and-error as problem-solving approach, and
- described their interactions with the computer as a partnership.

Moreover, they invariably indicated concrete or visual examples when describing successful instructional strategies. There was no evidence, however, that the contextual learners regularly eschewed planning, an implication that might be drawn from Turkle and Papert's (1990) choice of the term "planner" to describe people whose style was analytical.

Amy, the eighth participant, appeared at different times to demonstrate characteristics of both the analytical and contextual problem-solving styles. Furthermore, she seemed able to move between the two styles as needed.

Contextual learners. The students with low IPDT scores, those who in Piagetian theory would be classified as concrete learners, uniformly exhibited a contextual learning style. Excluding performance that could be attributed to fundamental misconceptions, the contextual learners were not as successful as the analytical learners. If creating a more inclusive computing discipline is a genuine goal, then computer programming educators must consider strategies that will better serve the group of students whose problem-solving style is contextual. Moreover, given the regularity with which the analytical learners referred to the value of the concrete representations and other strategies believed to assist the contextual learner and the difficulties displayed by most participants, it is reasonable to hypothesize that analytical learners would benefit from their employment as well.

According to Turkle and Papert (1990), the contextual learner is likely to be initially uncomfortable with the idea of procedural abstraction (of which parameter passing is an integral component), preferring to stay in contact with the program's details. While procedural abstraction is an essential aspect of the structured programming paradigm to which programmers must become accustomed, the transition can be facilitated by introducing the concept with a concrete, visual programming language. Karel the Robot (Pattis, 1981) and Logo's turtle graphics are two languages that have been used successfully to accomplish this task (personal communication, J. Kmoch, September 1993). As the designated languages can function without data, the introduction of parameters can be postponed until novices are acclimated to procedure use. Introducing parameters with graphics examples where students can observe the parameter effects (see the Banner program, for example, in Appendix F) is likely helpful, and algorithm visualization and animation techniques have also been shown to be successful.

Given their reactions to the abstract interview tasks, contextual learners may be more successful if the problems they are asked to solve are placed in an understandable context. Moreover, they would likely benefit from increased attention to the transparent, visual copy versus shared memory-location perspective of the parameter process.

At the semester's end, most participants still seemed to be consolidating fragile knowledge and naive conceptions of the parameter construct. Restructuring the course to introduce the procedure topic immediately would undoubtedly allow more time for greater attention to the troublesome area.

Beyond the immediacy of the present Pascal programming environment, other options exist. In recent years, for example, the movement to replace structured programming with an object-oriented paradigm has been gaining momentum. That paradigm may foster an environment that is more agreeable to the concrete thinker. Turkle and Papert (1990) observed:

> In the traditional concept of a program, the unit of thought is an instruction to the computer to do something. In object-oriented programming the unit of thought is creating and modifying interactive agents within a program (p. 155).

They additionally speculated that the growing support for icons may be part of a growing acceptance of concrete ways of thinking.

Where it is feasible (or possible) to identify a learner's preferred problem-solving style before he or she enrolls in a programming course, instruction could be adapted accordingly. It may be, for example, that a sequence of programming courses beginning with an assembly language experience--where the programmer is continually immersed in the details of a program--would be more congenial to the contextual learner.

Summary. The participants in the study demonstrated their principled understanding of the parameter construct in a manner that reflected the CIS 110 instruction, generally embracing the communication-oriented view. As that view allowed the development of a flawed variable parameter model, it was concluded that the communication view would be more useful if it was integrated with the shared memory-location view.

The eight participants displayed diverse problem-solving styles that generally reflected the styles reported in Turkle and Papert (1990). After allowing for misconceptions, it appeared that those with an analytical style were more successful than those with a contextual style. As the problem statement expressed concern over the success and retention rates of struggling novice programmers, several suggestions for enhancing the programming experiences of the contextual learners (and perhaps all learners) were offered.

Conceptual Understanding and Procedural Knowledge

What is the relationship between a novice's conceptual understanding and procedural knowledge of parameter passing? For this study, conceptual understanding was operationally defined as understanding of the parameter construct and the behavior of programs employing the construct, while procedural knowledge was defined as skill in constructing modular programs.

Without exception, the procedural skills of the participants appeared to surpass or equal their conceptual understanding of the construct. All participating students except Jason submitted every assignment, and the programs generally produced the correct results. All participants successfully completed the computer-assisted modification task; with the exception of Steve, they seemed to manage the task easily. Furthermore, excluding difficulties that could be attributed to a fundamental misunderstanding of the parameter concept, the participants appeared to have little difficulty with the hand-construction task. There are several possible reasons why this is so.

Goals of an introductory programming course. Perhaps the most persuasive explanation for the earlier development of procedural skills is that the introductory programming course, as it is traditionally taught, is a skills development course. Programmers are ultimately judged by the skill with which they construct programs; accordingly, instruction has been designed to develop that skill. The course in which the participants were enrolled was no exception. The textbook and classroom instruction stressed procedural knowledge heavily, and a review of the student activities and test questions confirms that the assessment matched the goals. As a result, the students had less experience with analysis and understanding tasks than they did with construction tasks.

This explanation for the uneven development of conceptual understanding and construction skills suggests that perhaps the parameter-related goals of the introductory course should be redefined to focus more on the former, and less on the latter. The literature does provide some support for such a re-definition. More than a decade ago, Deimel et al. (1983) expressed surprise concerning the continued write-only tradition of computer science. Van Merrienboer and his associates (1987, 1990, 1992) investigated strategies that could sustain such a re-definition. They advocated that novices read programs and complete partially constructed programs, contending that the student must understand the program to change it or complete it. The Clancy and Linn (1992) case study approach requires students to answer questions about worked-out code, maintaining that such activities foster understanding. Both schemes show promise. The students' success during the semester in question was at least partly attributed to activities of precisely the type proposed by Van Merrienboer and his associates, and the interview results, in which students appeared to construct their understanding as they introspected, seem to support to Clancy and Linn's work.

Regarding the goal of the parameter instruction in an introductory course, it appears the programming education community has two options. It can continue to stress success and procedural skills (what the student can do). Attendant with that choice is the likelihood that many students will continue to acquire only limited understanding of the parameter construct during their introductory course. Conversely, the community can shift the emphasis to conceptual knowledge (what the student understands). Attendant with that choice is the possibility that students will acquire less procedural skill during their introductory course.

Obscured Misconceptions. A second explanation for the conceptual-procedural relationship proposes that students with significant parameter misconceptions can construct programs that produce the correct answer by making seemingly innocuous adjustments to the parameter lists. The study provided compelling evidence that students whose understanding was fragile a) overused variable parameters, b) used identical actual and formal parameter names exclusively, or c) consistently provided identical formal parameter names when different modules referred to the same calling-module variable. In various ways, the adjustments allowed students to construct correctly working programs despite fragile understandings and significant misconceptions. The adjustments, however, concealed from the instructor, and perhaps from the students themselves, their misunderstanding.

Computer as Coach. Beyond the preceding explanations is the fact that the students have the computer to act as the expert, or more able peer, when they are constructing programs (Pea, 1985). The Pascal compiler (the program that converts Pascal statements into machine language) finds syntax (typographical, punctuation, and spelling, for example) errors, and forces students to confront the language-associated mistakes they make when composing programs. Programs do not compile until they are syntactically correct. Novice programmers, and perhaps all programmers, rely on the compiler's assistance to reduce the information processing load associated with constructing computer programs. The participants in this study were no exception. For example, some students required several attempts before successfully accomplishing the relatively simple (compared to other class-related construction tasks) task of modularizing the one-module program.

Perhaps more importantly, the computer provides students with concrete representations of the solutions they construct. The students know--or can know--when their solutions are correct. There is no doubt that programmers, not only novice programmers, rely on this assistance. Again, the students in this study were no exception. Several mentioned during the hand-construction task that they would be more comfortable if they had the computer to help them find their errors. Moreover, the computer-assisted task was the only one that led to eventual success for all participants.

Besides the general assistance described above, the machine provides specific assistance to students constructing modular programs. When executed, the program produces the wrong results if the programmer uses value parameters when variable parameters are needed. Amy, for example, recognized immediately that she needed to change value parameters to variable parameters when her modularized version of the payroll program reported zero results. On the other hand, substituting a variable parameter for a more appropriate value parameter does not yield erroneous results unless the program has other errors. There is some evidence that participants relied on this fact. Amy, Sammy, Steve, and Moe all inappropriately used variable parameters during the interview tasks and in some of their class work, and Steve forthrightly admitted that he gave a display procedure variable parameters “just to be on the safe side.”

Failure to correctly make value-variable choices (which almost invariably meant the student inappropriately chose variable parameters) was in truth the most commonly observed mistake in the participants' interview task performance, and in their class work. The most benign interpretation of the overuse of variable parameters is that most students had difficulty determining when they needed a value parameter and when they needed a variable parameter.

It may be that the computer-provided assistance, in the form of answers, relieved some students of the need for the mindful abstraction that Van Merrienboer and his associates (1987, 1990, 1992) deemed so important. It is possible that, when uncertain of the value-variable choice, some students had discovered that they could construct programs that produced the correct results by using variable parameters. If students equate success with a program that produces correct results (as the interview comments seemed to imply), and if modular programs that rely on variable parameters more often produce the correct results, then the computer-produced feedback of their solutions perhaps relieved some students of the need to carefully consider the value-variable parameter choices. Assuming this interpretation has merit, then an instructor's sole or heavy reliance on computer-assisted, construction-only parameter assignments could inhibit the development of students' conceptual understanding of the construct.

As encouraging novices to carefully attend to value-variable choices would seem to enhance both understanding and construction skill, several suggestions are provided.

Whether the suggestions have value, however, will not be known until they are tested empirically.

Enriched with questions requiring explanations, the focused completion and modification laboratory exercises (see Appendix E) reported in this study show promise. Such exercises would be consonant with strategies suggested by Van Merrienboer and his associates (1987, 1990, 1992) and Linn and Clancy (1990, 1992). A second possibility is the increased use of tasks such as the hand-construction task, perhaps enriched with the requirement that students explain the reasons for their choices. A third suggestion is the employment of computer-assisted assignments with the sole criteria of correctly constructed procedure heading lines.

Still another instructional strategy might be to focus attention on the appearance of the formal parameter lists, both in the teacher-provided models and in the student work. For example, instructors might insist that parameters be listed vertically (one per line), that all value parameters appear before any variable parameters, or that multiple parameters not be combined in lists separated by commas. Close attention to the appearance of the formal parameter list might induce students to attend more closely to their value-variable decisions. An experimental investigation of the impact of various forms of the parameter list might yield helpful information.

Additionally, the topic seems an ideal candidate for classroom debates or small-group work. Such activities would provide novices opportunities for collaboration, elaboration, conflict resolution, and construction of their own criteria for correctness.

Abstract Tasks. Subordinate to the preceding explanations for the conceptual-procedural relationship is one related to the nature of the interview tasks. The construction tasks were familiar and contextually rich, while the analysis tasks were abstract. It is certain the abstract quality affected some participants' performance.

Summary. The participants in the study demonstrated procedural skills that exceeded their conceptual understanding of the parameter construct. Several explanations have been proffered to explain the conclusion. First, an emphasis on skill development is consonant with the traditional goals of the course. Second, students could screen fundamental misconceptions and still write routine programs that would produce the correct results by making relatively minor adjustments to the program's parameters. Third, the computer helps find errors when a program is constructed incorrectly. Moreover, it is forgiving of inappropriately used variable parameters. Finally, the interview analysis tasks were abstract in contrast to the contextually-rich construction tasks.

Success

What impact does a student's understanding of the parameter passing construct have on his or her eventual success in an introductory programming course?

Measured by the traditional standard (the student's final grade), the students enrolled in the research section were very successful. Thirty students completed the surveys at the semester's onset; 26 completed the course. Of those, 10 earned A's, 7 earned A-'s, and 9 earned B+'s; thus, all students who completed the tests and submitted the assignments earned a grade of at least B+. A generous grading scale may account for the high success rate. Conversely, much about the course changed dramatically in the year preceding the study; it is likely those changes contributed to the high success and retention rates.

As was the class overall, the eight participants were inordinately successful. The final grade pattern, provided in Table 3, does seem to mildly suggest that those students whose parameter understanding seemed most principled were the most successful. The tight cluster of grades, however, prohibits drawing any but the most tentative conclusions.

Table 3

Participants' final grades

Name	Carol	Carl	Amy	Sammy	Sondra	Steve	Moe	Jason
Grade	A	A	A	B+	A-	A-	A-	A-

Carol and Carl, for example, consistently and relatively effortlessly accomplished the interview tasks, and offered explanations that demonstrated understanding. Amy succeeded with all but one of the tasks, albeit after a bit of struggle. However, the success of the others, especially those who appeared to harbor fundamental parameter misconceptions, raises doubts.

The problem statement asserted that understanding the parameter construct is crucial for sucess. As participants who appeared to demonstrate only limited understanding of the construct or to harbor serious misconceptions were successful, the study did not support the problem statement. Therefore, it is fitting to attempt to briefly account for the participants'success.

First, the parameter construct is only one of several studied throughout the span of the semester. While the parameter construct was central to one mid-term examination, a perusal of the related test questions reveals that few focused on understanding. Moreover, most of the parameter assignments were program construction activities. Students with fragile understanding could construct programs that produced the correct answer by making seemingly innocuous adjustments to parameter lists. Overuse of variable parameters also allowed students to construct correctly-working programs without fully understanding when variable parameters were the appropriate choice.

There is some reason to hypothesize that the addition of the structured laboratory component to the course positively impacted student performance. The overall success of the class, the positive reaction of most students, and the instructor's observations appear to support the claims of benefit revealed in the literature. Two aspects of the structured laboratory experience emerged as significant: the collaboration it fostered and infusion of focused, completion-type programming exercises.

Expressing opinions consistent with the literature, the instructor and several of the participants--especially some of those who struggled--attested to the value of the collaboration that was promoted by the structured laboratory environment. Moreover, the nature of the informal, self-selected student groups was consistent with the successful grouping patterns reported in Webb and Lewis (1988). Furthermore, the collaboration reported in this study mirrored the literature reports of students working in communities of practice that are, according to Collins et al., "fostered by common projects and shared experiences" (1991, p. 45).

The instructor and several of the students attributed the enhanced achievement partly to the incorporation of focused, completion-type parameter exercises. The perception of benefit supports the findings of Van Merrionboer and Paas (1990) and Van Merrionboer and De Crook (1992) reported earlier in the paper. At this time, the claim of superiority for the completion-type parameter exercises over construction-type exercises is an untested hypothesis. Whether the hypothesis is borne out remains for further investigation.

Effect of Understanding on Disposition

What impact does a student's understanding of the parameter passing construct have on his or her disposition toward computer programming? The study provided limited evidence that improved understanding of the parameter construct could advance a more favorable disposition toward computer programming. The participants commonly used the term "challenging" to describe the introductory programming course, and they unanimously agreed that the parameter construct was the most challenging concept the course entailed. However, they seemed to welcome the challenge as long as they were generally successful. As improved understanding would likely result in more successful programming experiences (as would improved understanding of any other programming construct), improved understanding of the parameter construct could indirectly help to generate a more positive disposition toward computer programming.

Findings and Implications of the Study

Several conclusions emerged from the interpretation of the individual cases and the preceding analysis. A discussion of the study's major findings and the implications of the study follows.

Finding 1

Students' understanding of the natural-language meaning of some terminology appeared to interfere with their adoption of the technical meaning of the terms, possibly inhibiting their understanding of the parameter construct. The term "variable" seemed especially suspect. Variable is a fundamental programming construct, and it was covered early during the instruction. Moreover, most participants reported having some programming experience where they would have previously encountered the concept. The finding that students may have lacked an understanding of the data storage component of the term has several implications. It implies first that instructors cannot assume novices attach the same meaning to the terminology that experts do. Moreover, it suggests that instructors cannot assume students understand a concept simply because they have been previously exposed to it, regardless of the quality of prior instruction. Instruction on crucial ideas, consequently, should be spiral, and teachers should structure experiences that require learners to attend to both the essential and the abstruse aspects of a concept. In a more global sense, the finding counsels that textbook authors, publishers of instructional materials, compiler writers, and inventors of new programming languages should be sensitive to the difficulties inherent in some terminology.

Finding 2

Most students displayed either fragile understanding of the parameter construct, or solved problems in a manner consistent with a systematic application of a fundamental misconception of the parameter process. Both analytical learners and contextual learners experienced misconceptions.

It was apparent that most participants in the study were still constructing or consolidating a fragile understanding of the parameter construct several weeks after the conclusion of the instruction. The finding suggests that perhaps the procedure-parameter topic should be introduced very early in the learning experience to provide students with a greater number and variety of parameter learning activities, and more time to construct their understanding.

The most pervasive instance of fragile understanding was the participants' uncertainty regarding their value-variable parameter choices. To encourage the development of decision-making skill, learners need to be provided with carefully structured situations requiring them to construct their own criteria for the correctness of their value-variable decisions.

Among the eight participants, two analytical learners and one contextual learner demonstrated problem-solving behaviors suggestive of fundamental misconceptions of the parameter construct. Consequently, instructors cannot assume that most students construct an undistorted image of the parameter process during initial instruction. Furthermore, they cannot assume that contextual learners are more likely than analytical learners to develop flawed conceptions. Importantly, all three students who harbored misconceptions verbalized their inability to understand. This fact suggests that students can provide helpful information regarding their conceptions and misconceptions if they are encouraged to do so.

Consistent with the literature on misconceptions, some oral and written responses that initially appeared capricious, and even bizarre, proved to be logical and appropriate choices when interpreted in terms of the learner's hypothesized misconception. The finding, thus, suggests that programming teachers refrain from interpreting program errors as simply carelessness or lack of attention or lack of concern on the student's part.

Finding 3

The mental model of the parameter process constructed by the students reflected the conceptual model provided by the classroom and textbook parameter instruction. The textbook and classroom instruction stressed the one-way versus two-way communication model, and the students universally adopted the communication-oriented mental model of the process. Guided by the opaque communication model, all participants appeared to imagine some variation of a mechanism whereby the value in a formal variable parameter replaced the value in the corresponding actual parameter at the conclusion of a procedure's execution.

Given that the replacement idea is a flawed image of the variable parameter process, it is possible to hypothesize that increased emphasis on the more transparent shared memory-location perspective would allow novices to construct a more authentic mental model of the parameter process. As the incorporation of concrete representations seemed to be a powerful strategy for assisting students in constructing their mental models, the provision of a concrete representation of the shared memory-location perspective would appear to be beneficial. Beyond that, the finding counsels instructors to remember that learners do not automatically construct understandings that propel them toward a principled understanding of a concept. Learning situations must be carefully structured if they are to advance the desired understandings, those compatible with expert perceptions.

Finding 4

Students with an analytical problem-solving style generally demonstrated a greater understanding of the parameter construct than those with a contextual problem-solving style. This finding may be attributed in part to portions of the instruction (the opaque, communication-oriented parameter model) and to the abstract nature of some interview and assessment tasks, which possibly hindered learners with a contextual problem-solving style and favored those with an analytical style. The conclusion implies that programming instruction must provide a climate responsive to students' different learning styles by providing a broad array of examples and activities that would allow students with varied learning styles to benefit from the class.

Finding 5

Students' procedural knowledge appeared to equal or surpass their conceptual understanding of the parameter construct. The finding can be attributed at least in part to the traditional skill-oriented goals of computer programming courses. Given the students' relative success with procedural skills and the nature and extent of the conceptual difficulties disclosed by the study, a re-definition of the goals to focus more on understanding and less on skill development (at least for the parameter construct) seems reasonable.

Finding 6

Programs that produce the correct answers can camouflage a student's lack of understanding from him or herself and from the instructor. The study disclosed that apparently innocuous adjustments to parameter names and value-variable choices allowed students to construct programs that produce the correct results despite fundamental misunderstandings of the parameter construct. In this manner, students' lack of understanding was camouflaged from the instructor and perhaps from the learners themselves. The study also revealed that programming students value success and that, moreover, they equate correctly working programs (programs that produce the correct results) with success. Hence, the students presumably employed whatever strategies were necessary to achieve success. If understanding of the programming language constructs is a legitimate and important goal of programming instruction, the findings suggest that instructors of introductory programming courses must structure opportunities whereby learners construct criteria for programming success that are more comprehensive than merely "finding the right answer." Redefined goals that more heavily emphasize conceptual understanding would likely facilitate the incorporation of such activities.

Finding 7

Both analytical learners and contextual learners were generally successful in the course and in their ability to construct original modular programs. The conclusion implies that computer programming educators cannot assume a student will be unsuccessful solely because his or her predominant learning or problem-solving style differs from that traditionally associated with successful computer programmers. They must, instead, assume *a priori* that every student has the potential to succeed. Moreover, combined with other conclusions, the finding suggests that instructors cannot assume that students understand the parameter construct simply because they are able to construct programs that employ the concept.

Finding 8

The interview process provided an instructional intervention when participants explained their reasoning. The impact appeared to be most notable for participants whose understanding was fragile. Mayer (1982) proposed that encouraging a learner to explain in his or her own words and to relate new information to other ideas or concepts (to elaborate) supports the activation of relevant existing knowledge, and assists in the assimilation of new material. The participants in this study who elaborated during the interviews appeared to reap the benefit proposed by Mayer. This finding strongly suggests that providing students with opportunities to elaborate will enhance their understanding of the parameter construct.

Finding 9

Concrete representations, focused parameter activities, and a structured laboratory environment that promoted collaboration appeared to contribute to students' success. The finding implies that the continued and expanded use of these instructional techniques could enhance students' ability to understand and use the parameter construct.

Limitations of the Study and Future Directions

The study is subject to all the limitations inherent in a qualitative study (Merriam, 1988). Moreover, of the instruments and protocols employed, only the IPDT had undergone the rigor of validity and reliability testing (Milakofsky & Patterson, 1979). With the exception of the second analysis task (Task 4, Appendix P) excerpted from the Advanced Placement Course Description (College Board, 1993, p. 41), the tasks utilized in the exploratory study were the invention of the researcher. They reflect one instructor's view of skills and understandings that might be important to assess when studying novice Pascal programmers learning the parameter passing construct.

Owing to the exploratory nature of the study, some tasks may not have probed participants' understanding as deeply as possible. Given, for example, the unexpected emergence of the direct procedure-to-procedure link misconceptions, the analysis tasks (those incorporating only one procedure) likely yielded less information regarding the novices' intuitive conceptions than would tasks with multiple procedures. Moreover, no tasks included variables superfluous to the parameter process; the likelihood of correct guessing was thereby increased. Additionally, no task required the programmer to construct actual parameter lists to match existing formal parameter lists. Attention to these limitations provides guidance for future researchers.

While the researcher was not the instructor of the studied class, she was actively involved in developing the laboratory exercises, the original programming assignments, and the examination questions. There is, as a result, a congruence between the interview tasks and the instruction that may not have existed had the researcher been less actively involved in the instruction. Different results would likely come out of a research condition in which this were not the case.

Finally, the researcher's bias must not be minimized. The researcher in this study subscribes to the position of Turkle and Papert (1990) that concrete (contextual) and formal (analytical) problem-solving approaches are both productive styles of thinking. A researcher who regarded them as stages of development might interpret the case study data differently.

Bibliography

AAUW. (1992). How schools shortchange girls: A study of major findings on girls and education. American Association of University Women.

Alspaugh, C. A. (1972). Identification of some components of computer programming aptitude. Journal for Research in Mathematics Education, 3, 89-98.

Anderson, J. R. (1983). The architecture of cognition. Cambridge, MA: Harvard University Press.

Armstrong, A. M. (1989). The development of self-regulation skills through the modeling and structuring of computer programming. Educational Technology Research and Development, 37(2), 69-76.

Bastian, S., Frees, J., Gruber, Sr. L., Johnson, J., Landes, B., Morton, L., Rozgony, S., & Stewart, J. (1973). Are ISU freshmen students operating at a formal level in thought processes? Contemporary Education, 44, 358-362.

Battista, M. T. (1990). Spatial visualization and gender differences in high school geometry. Journal for Research in Mathematics Education, 21(1), 47-60.

Battista, M. T. & Clements, D. H. (1991). Using spatial imagery in geometric reasoning. Arithmetic Teacher, 39(1), 18-22.

Bauer, R., Mehrens, W. A., & Vinsonhaler, J. F. (1968). Predicting performance in a computer programming course. Educational and Psychological Measurement, 28, 1159-1164.

Beaulieu, J. E. (1990). Using structure charts to teach problem solving and program design in beginning Pascal. Journal of Computer Science Education, 5(1), 13-20.

Beilin, H. (1992). Piaget's enduring contribution to developmental psychology. Developmental Psychology, 28, 191-204.

Belmont, J. M. (1989). Cognitive strategies and strategic learning: The socio-instructional approach. American Psychologist, 44, 142-148.

Bonar, J. & Soloway, E. (1985). Preprogramming knowledge: A major source of misconceptions in novice programmers. Human-Computer Interaction, 1, 133-161.

Booth, S. (1990). Conceptions of programming: A study of learning to program. (ERIC Document Service No. ED 338 227).

Borman, K. M., Le Compte, M. D., & Goetz, J. P. (1986). Ethnographic and qualitative research design and why it doesn't work. American Behavioral Scientist, 30(1), 42-57.

Brooks, R. (1977). Towards a theory of the cognitive processes in computer programming. International Journal of Man-Machine Studies, 9, 737-751.

Brown, C., Fell, H., Proulx, V. K., & Rasala, R. (1992). Using visual feedback and model programs in introductory computer science. Journal of Computing in Higher Education, 4(1), 3-26.

Bruner, J. (1960). The process of education. Cambridge, MA: Harvard University Press.

Bruner, J. (1966). Toward a theory of instruction. Cambridge, MA: Harvard University Press.

Bruner, J. (1973). The relevance of education. New York: Norton.

Burton, J. K. & Magliaro, S. (1987-88). Computer programming and generalized problem-solving skills: In search of direction. Computers in the Schools, 4(3-4), 63-90.

Cafolla, R. (1987-1988). Piagetian formal operations and other cognitive correlates of achievement in computer programming. Journal of Educational Technology Systems, 16, 45-55.

Cavaiani, T. P. (1989). Cognitive style and diagnostic skills of student programmers. Journal of Research on Computing in Education, 21, 411-420.

Cheny, P. (1980). Cognitive style and student programming ability: An investigation. AEDS Journal, 13, 285-291.

Choi, W. S. & Repman, J. (1993). The effects of Pascal and FORTRAN programming on the problem-solving abilities of college students. Journal of Research on Computing in Education, 25, 290-302.

Chung, C. (1988). Correlates of problem solving in programming. CHUK Education Journal, 16, 185-190.

Clancy, M. J. & Linn, M. C. (1992a). Case studies in the classroom. SIGCSE Bulletin, 24, 220-224.

Clancy, M. J. & Linn, M. C. (1992b). Designing Pascal solutions: A case study approach. New York: Freeman.

Clement, C. A., Kurland, D. M., Mawby, R., & Pea, R. D. (1986). Analogical reasoning and computer programming. Journal of Educational Computing Research, 2, 473-485.

College Board. (1993). Advanced placement course description: Computer science. USA: College Entrance Examination Board.

Collins, A., Brown, J. S., & Holum, A. (1991). Cognitive apprenticeship: Making thinking visible. American Educator, 15(3), 6-11, 38-46.

Cooper, D. (1985). Teaching introductory programming. New York: Norton.

Cooper, D. & Clancy, M. C. (1985). Oh! Pascal! New York: Norton.

Dalbey, J. C. & Linn, M. C. (1985). The demands and requirements of computer programming: A literature review. Journal of Educational Computing Research, 1, 253-274.

Dalbey, J., Tournaire, F., & Linn, M. (1986). Making programming instruction cognitively demanding: An intervention study. Journal of Research in Science Teaching, 23, 427-436.

Deimel, L. E., Jr., Hodges, L. F., & Moffat, D. V. (1983). Restructuring the introductory programming course. AEDS Monitor, 21(7-8), 11-15.

Du Boulay, B. (1986). Some difficulties of learning to program. Journal of Educational Computing Research, 2, 57-73.

Du Boulay, B., O'Shea, T., & Monk, J. (1989). The black box inside the glass box: Presenting computing concepts to novices. In E. Soloway & J. C. Spohrer (Eds.), Understanding the novice programmer (pp. 431-446). Hillsdale, NJ: Lawrence Erlbaum.

Ericsson, K. A. & Simon, H. A. (1980). Verbal reports as data. Psychological Review, 87, 215-251.

Eylon, B. & Linn, M. (1988). Learning and instruction: An examination of four research perspectives in science education. Review of Educational Research, 58, 251-301.

Fay, A. L. & Mayer, R. E. (1988). Learning LOGO: A cognitive analysis. In R. E. Mayer (Ed.), Teaching and learning computer programming: Multiple research perspectives (pp. 55-74). Hillsdale, NJ: Lawrence Erlbaum.

Fischer, G. B. (1986). Computer programming: A formal operational task. Paper Presented at the 16th Annual Symposium of the Piaget Society. (ERIC Document Service No. ED 275 316).

Foreman, K. H. (1990). Cognitive characteristics and initial acquisition of computer programming competence. School of Education Review, 2(Spring), 55-61.

Fowler, G. C. & Glorfeld, L. W. (1981). Predicting aptitude in introductory computing: A classification model. AEDS Journal, 14, 96-109.

Furth, H. (1970). An inventory of Piaget's developmental tasks. Center for Research in Thinking and Language, Department of Psychology, Catholic University: Washington, D. C.

Gagné, E. D., Yekovich, C. W., & Yekovich, F. R. (1993). The cognitive psychology of school learning. New York: HarperCollins.

Geisert, G. & Dunn, R. (1991). Effective use of computers: Assignments based on individual learning style. The Clearing House, March/April, 70-75.

Grandgenett, N. & Thompson, A. (1991). Effects of guided programming instruction on the transfer of analogical reasoning. Journal of Educational Computing Research, 7, 293-308.

Greer, J. (1986). High school experience and computer achievement in computer science, AEDS Journal, 19, 216-225.

Hancock, C. (1988). Context and creation in the learning of computer programming. For the Learning of Mathematics, 8(1), 18-24.

Hatch, J. A. (1985). The quantoids versus the smooshes: Struggling with methodological rapprochement. Issues in Education, 3, 158-167.

Hayek, L. M. & Stephens, L. (1989). Factors affecting computer anxiety in high school computer science students. Journal of Computers in Mathematics and Science Teaching, 8(4), 73-76.

Hostetler, T. R. (1983). Predicting student success in an introductory programming course. SIGCSE Bulletin, 15(3), 40-49.

Howell, K. (1993). The experience of women in undergraduate computer science: What does the research say? SIGCSE Bulletin, 25(2), 1-7.

Inhelder, B. & Piaget, J. (1958). The growth of logical thinking from childhood to adolescence. New York: Basic Books.

Inhelder, B. & Piaget, J. (1980). Procedures and structures. In D. R. Olson (Ed.), The social foundations of language and thought. New York: Norton.

Johnson, D. W. & Johnson, R. T. (1986). Computer-assisted cooperative learning. Educational Technology, 26(1), 12-18.

Johnson, J. A. & Johnson, G. M. (1992). Student characteristics and computer programming competency: A correlational analysis. Journal of Studies in Technical Careers, 14(1), 33-46.

Johnson, R. E. & Keil, D. M. (1995). Introduction to computer programming using Turbo Pascal. St. Paul: West.

Joni, S. A. & Soloway, E. (1986). But my program runs! Discourse rules for novice programmers. Journal of Educational Computing Research, 2, 95-125.

Kagan, D. M. (1988). Learning how to program or use computers: A review of six applied studies. Educational Technology, 28(3), 49-51.

Knudson, D. L. (1987). Teaching complexity of algorithms using concurrent real-time graphic animation of various sorting techniques. Proceeding of the 20th Annual Small College Computing Symposium, 217-227.

Knudson, D. (1988). Sort-Demo [Computer program]. LaCrosse, WI.

Kong, S. C. & Chung, C. M. (1989). Effects of language features, templates, and procedural skills on problem-solving in programming. CHUK Education Journal, 27, 79-88.

Koubek, R. J., Le Bold, W. K., & Salvendy, G. (1985). Predicting performance in computer programming courses. Behaviour and Information Technology, 4, 113-129.

Kuhn, D., Ho, V., & Adams, C. (1979). Formal reasoning among pre- and late adolescents. Child Development, 50, 1128-1135.

Kull, J. A. (1988). Children learning Logo: A collaborative, qualitative study in the first grade. Journal of Research in Childhood Education, 3, 55-75.

Kurland, D. M., Pea, R. D., Clement, C., & Mawby, R. (1986). A study of the development of programming ability and thinking skills in high school students. Journal of Educational Computing Research, 2, 429-458.

Kurtz, B. L. (1980). Investigating the relationship between the development of abstract reasoning and performance in an introductory programming class. ACM SIGCSE Bulletin, 110-117.

Kwan, S. K., Trauth, E. M., and Driehaus, K. C. (1985). Differences and computing: Students' assessment of societal influences. Education and Computing, 1, 187-194.

Lawson, A. E. (1985). A review of research on formal erasing and science teaching. Journal of Research in Science Teaching, 22, 569-617.

Lawson, A. E., Mc Elrath, C. B., Burton, M. S., James, B. D., Doyle, R. P., Woodward, S. L., Kellerman, L., & Snyder, J. D. (1991). Hypothetico-deductive reasoning skill and concept acquisition: Testing a constructivist hypothesis. Journal of Research in Science Teaching, 28, 953-970.

Leeper, R. R. & Silver, J. L. (1982). Predicting success in a first programming course. SIGCSE Bulletin, 14, 147-150.

Leestma, S. & Nyhoff, L. (1993). Turbo Pascal: Programming and problem solving (2nd ed.). New York: MacMillan.

Lemerise, T. (1993). Piaget, Vigotsky, & Logo. The Computing Teacher, 23(7), 24-28.

Lincoln, V. S. & Guba, E. G. (1985). Naturalistic inquiry. Beverly Hills, CA: Sage Productions.

Linn, M. C. (1985). The cognitive consequences of programming instruction in classrooms. Educational Researcher, 14 (5), 14-16, 25-29.

Linn, M. C. (1992). How can hypermedia tools help teach programming? Learning and Instruction, 2, 119-139.

Linn, M. C. & Clancy, M. J. (1990). Designing instruction to take advantage of recent advances in understanding cognition. Academic Computing, April, 20-23, 35-41.

Linn, M. C. & Clancy, M. J. (1992). The case for case studies of programming problems. Communications of the ACM, 35, 121-132.

Linn, M. C. & Dalbey, J. (1985). Cognitive consequences of programming instruction: Instruction, access, and ability. Educational Psychologist, 20, 191-206.

Linn, M. C. & Songer, N. B. (1991). Cognitive and conceptual change during adolescence. American Journal of Education, 99, 379-417.

Linn, M. C., Sloane, K. D., & Clancy, M. J. (1987). Ideal and actual outcomes from precollege Pascal instruction. Journal of Research in Science Teaching, 24, 467-490.

Lockheed, M. E. & Mandinach, E. B. (1986). Trends in educational computing and the changing focus of instruction. Educational Researcher, 15(5), 21-26.

Madison, S. & Gau, G. E. (1993). Formal thinking and computing students. Proceedings of the Midwest Computer Conference, 55-62.

Mandinach, E. B. & Linn, M. C. (1986). The cognitive effects of computer learning environments. Journal of Educational Computing Research, 2, 411-427.

Marchionini, G. (1985). Teaching programming: A developmental approach. The Computing Teacher, 12(9), 12-15.

Mawhinney, C. H., Cale, E. G., Jr., & Callaghan, D. R. (1990). Freshmen expectations of information systems careers versus their own careers. CIS Educator Forum, 2(3), 2-8.

Mayer, R. E. (1982). Diagnosis and remediation of computer programming skill for creative problem solving. Volume 1: Description of research methods and results. Final Report. Santa Barbara, California: University of California. (ERIC Document Services No. ED 230 199).

Mayer, R. E. (1985). Learning in complex domains. In G. H. Bower (Ed.), The psychology of learning and motivation (pp. 89-130). Orlando, FL: Academic Press.

Mayer, R. E. (1988). Teaching and learning computer programming: Multiple research perspectives. Hillsdale, NJ: Lawrence Erlbaum.

Mayer, R. E. (1989). The psychology of how novices learn computer programming. In E. Soloway & J. C. Spohrer (Eds.), Understanding the novice programmer (pp. 129-159). Hillsdale, NJ: Lawrence Erlbaum.

Mazlak, L. J. (1980). Identifying potential to acquire programming skill. Communications of the ACM, 23(1), 14-17.

McGrath, D. (1990). Using mental models to teach programming to beginners. The Computing Teacher, 17(9), 11-14.

McIntyre, P. (1991). Teaching structured programming in the secondary schools. Malabar, FL: Krieger.

McKeithen, K. B., Reitman, J. S., Reuter, H. H., & Hirtle, S. C. (1981). Knowledge organization and skill differences in computer programmers. Cognitive Psychology, 13, 307-325.

Mercer, R. (1992). Problem solving and program implementation using Turbo Pascal. Wilsonville, OR: Franklin Beedle.

Merriam, S. H. (1988). Case study research in education: A qualitative approach. San Francisco: Jossey-Bass.

Merritt, S. M. (1992). ACM model high school computer science curriculum. ACM SIGCSE Bulletin, 24(1), 39. (Abstract).

Milakofsky, L. & Patterson, H. O. (1979). Chemical education and Piaget. Journal of Chemical Education, 56, 87-90.

Mitchell, A. & Lawson, E. (1988). Predicting genetics achievement in nonmajors in college biology. Journal of Research in Science Teaching, 25, 23-37.

Multiscope, Inc. (1991). Multiscope Debugger for Windows [Computer program]. Mountain View, CA.

Nachmias, R., Mioduser, D., & Chen, D. (1986). Variables: An obstacle to children leaning computer programming. Paper presented at the AERA. (ERIC Document Services No. ED 290 459).

Nance, D. W. & Naps, T. L. (1989). Introduction to computer science: Programming, problem solving, and data structures. St. Paul: West Publishing.

Naps, T. L. (1989). Design of a completely general algorithm visualization system. Proceedings of the 22nd Annual Small College Computing Symposium, 233-241.

Naps, T. L. (1990a). Algorithm visualization in computer science laboratories. ACM SIGCSE Bulletin, 22, 105-110.

Naps, T. L. (1990b). Generalized Algorithm Illustration through Graphical Software (GAIGS) [Computer program]. Appleton, WI: Lawrence University.

Neuman, D. B. (1993). Experiencing elementary science. Belmont, CA: Wadsworth.

Newstead, P. R. (1975). Grade and ability predictions in an introductory programming course. SIGCSE Bulletin, June, 87-91.

Norris, C., Jackson, L., & Poirot, J. (1992). The effect of computer science instruction on critical thinking and mental alertness. Journal of Research on Computing in Education, 24, 329-337.

Oman, P. W. (1986). Identifying students' characteristics influencing success in introductory computer science courses. AEDS Journal, 19, 226-233.

Papert, S. (1980). Mindstorms. New York: Basic Books.

Patterson, H. O. & Milakofsky, L. A. (1980). A paper-and-pencil inventory for the assessment of Piaget's tasks. Applied Psychological Measurement, 341-353.

Pea, R. D. (1985). Integrating human and computer intelligence. In E. L. Klein (Ed.), Children and computers: New directions for child development. (pp. 75-95). San Francisco: Jossey-Bass.

Pea, R. D. (1986). Language-independent conceptual "bugs" in novice programming. Journal of Educational Computing Research, 2, 25-36.

Pea, R. D. (1983). On the cognitive prerequisites of learning computer programming. (Tech. Rep. No. 52). New York: Bank Street College of Education Center for Children and Technology.

Pea, R. D., Soloway, E., & Spohrer, J. C. (1987). The buggy path to the development of programming expertise. Focus on Learning Problems in Mathematics, 9(1), 5-30.

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1986). Conditions of learning in novice programmers. Journal of Educational Computing Research, 2, 37-55.

Perkins, D. N., Schwartz, S., & Simmons, R. (1988). Instructional strategies for the problems of novice programmers. In R. E. Mayer (Ed.), Teaching and learning computer programming: Multiple research perspectives (pp. 153-178). Hillsdale, NJ: Lawrence Erlbaum.

Perkins, D. N. & Simmons, R. (1988). Patterns of misunderstanding: An integrative model for science, math, and programming. Review of Educational Research, 58, 303-326.

Putnam, R. T., Sleeman, D., Baxter, J. A., & Kuspa, L. K. (1986). A summary of misconceptions of high school basic programmers. Journal of Educational Computing Research, 2, 459-472.

Rogers, J. (1982). Teaching beginners to program: Some cognitive considerations. (ERIC Document Service No. ED 219 880).

Sack, W., Soloway, E., & Weingrad, P. (1992). Re: Writing cartesian student models. Journal of Artificial Intelligence in Education, 3, 381-399.

Samurçay, R. (1985). Learning programming: An analysis of looping strategies used by beginning students. For the Learning of Mathematics, 5, 37-43.

Samurçay, R. (1989). The concept of variable in programming: Its meaning and use in problem-solving by novice programmers. In E. Soloway & J. C. Spohrer (Eds.), Understanding the novice programmer (pp. 161-178). Hillsdale, NJ: Lawrence Erlbaum.

Schneiderman, B. (1977). Teaching programming: A spiral approach to syntax and semantics. Computers and Education, 1, 193-197.

Shaklee, H. (1979). Bounded rationality and cognitive development: Upper limits on growth? Cognitive Psychology, 11, 327-345.

Sharma, S. (1986-87). Learners' cognitive styles and psychological types as intervening variables influencing performance in computer science courses. Journal of Educational Technology Systems, 15, 391-399.

Sheil, B. A. (1981). The psychological study of programming. Computing Surveys, 13, 101-120.

Simon, J. (1987). Vygotsky and the Vygotskians. American Journal of Education, 95, 609-613.

Sleeman, D., Putnam, R. T., Baxter, J., & Kuspa, L. (1988). An introductory Pascal class: A case study of errors. Journal of Educational Computing Research, 4, 5-23.

Sloane, K. D. & Linn, M. C. (1988). Instructional conditions in Pascal programming classes. In R. E. Mayer (Ed.), Teaching and learning computer programming: Multiple research perspectives (pp. 207-235). Hillsdale, NJ: Lawrence Erlbaum.

Smith, J. K. & Heshusius, L. (1986). Closing down the conversation: The end of the quantitative-qualitative debate. Educational Researcher, 15(1), 4-12.

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. Communications of the ACM, 29, 850-858.

Soloway, E. & Ehrlich, K. (1984). Empirical studies of programming knowledge. IEEE Transactions on Software Engineering, 10, 595-609.

Soloway, E. & Spohrer, J. C. (1989). Studying the novice programmer. Hillsdale, NJ: Lawrence Erlbaum.

Spohrer, J. C. & Soloway, E. (1986). Novice mistakes: Are the folk wisdoms correct? Communications of the ACM, 29, 624-632.

Stephens, L. J., Wileman, S., & Knovalina, J. (1981). Group differences in computer science aptitude. AEDS Journal, 14, 84-95.

Stevens, D. J. (1983). Cognitive processes and success of students in instructional computer courses. AEDS Journal, 16, 228-233.

Swan, K. (1993). Domain knowledge, cognitive styles, and problem solving: A qualitative study of student approaches to Logo programming. Journal of Computing in Childhood Education, 4, 153-182.

Taylor, R. P. & Cunniff, N. (1988). Moving computing and education beyond rhetoric. Teachers College Record, 89, 360-372.

Tharp, R. G. & Gallimore, R. (1989). Rousing schools to life. American Educator, 13(2), 20-25, 46-52.

Tuovinen, J. E. & Hill, D. M. (1992). Towards better strategies for thinking and programming. School Science and Mathematics, 92, 206-211.

Turkle, S. (1988). Computational reticence: Why women fear the intimate machine. In C. Kramarae (Ed.), Technology and women's voices: Keeping in touch. New York: Routledge and Kegan Paul.

Turkle, S. & Papert, S. (1990). Epistemological pluralism: Styles and cultures within the computer culture. Signs: Journal of Women in Culture and Society, 16, 128-148.

Van Merrienboer, J. J. G. (1988). Relationship between cognitive learning style and achievement in an introductory computer programming course. Journal of Research on Computing in Education, 21,181-186.

Van Merrienboer, J. J. G. & De Crook, M. B. M. (1992). Strategies for computer-based programming instruction: Program completion vs program generation. Journal of Educational Computing Research, 8, 365-394.

Van Merrienboer, J. J. G. & Krammer, H. P. (1987). Instructional strategies and tactics for the design of introductory computer programming courses in high school. Instructional Science, 16, 251-285.

Van Merrienboer, J. J. G. & Paas, F. G. W. C. (1990). Automation and schema acquisition in learning elementary computer programming: Implications for the design of practice. Computers in Human Behavior, 6, 273-289.

Webb, N. M. & Lewis, S. (1988). The social context of learning computer programming. In R. E. Mayer (Ed.), Teaching and learning computer programming: Multiple research perspectives (pp. 179-206). Hillsdale, NJ: Lawrence Erlbaum.

Wertsch, J. & Tulviste, P. (1992). L. S. Vygotsky and contemporary developmental psychology. Developmental Psychology, 28, 548-557.

Wileman, S., Konvalina, J., & Stephens, L. J. (1981). Factors influencing success in beginning computer science courses. Journal of Educational Research, 223-226.

Wresch, W. (1988). Cognitive barriers to initial programming instruction: A survey of research. Paper presented at National Educational Computing Conference.

Zirkler, D. & Brownell, G. (1991). Analogical reasoning: A cognitive consequence of programming? Computers in the Schools, 8, 135-145.

Appendix A

Instructor Informed Consent

As an assistant professor in the Department of Mathematics and Computing at the University of Wisconsin-Stevens Point. I am conducting a study of Pascal students' understanding of the construct of parameter passing, one of the pivotal concepts taught in CIS110. I would appreciate your participation in the study, as it may inform our efforts to improve the way we teach the construct.

Upon your consent and the students' consent, the students in section two of CIS110 which you teach will be given an Inventory of Piaget's Developmental Task (IPDT), a novel concepts task, and a background survey to complete. They take about 30 minutes total to complete and will be administered during the class. The students will be asked to retake the IPDT once more at the end of the semester. It will again be administered during class.

In addition, demographic and background information will be collected from student records. I will use the information to investigate possible relationships between these variables and the students' understanding of the construct of parameter passing.

I will observe selected sessions of the class during the semester and may videotape some activities.

Some students will be asked to participate in a series of three interviews as the class studies the construct of parameter passing. The interview questions will focus on the students' computing and programming background and understanding of the construct of parameter passing. The interviews will be audio tape recorded and transcribed. It is estimated that the interviews will last from 20-30 minutes. The interviews will be conducted outside of class. Interview candidates will be chosen to provide a broad range of prior computing experience and backgrounds.

I will also ask to interview you three times during the semester. The first interview will focus on your background, the second on the CIS110 class and the parameter construct in particular, and the third will ask you to reflect on those portions of the class.

There are no foreseeable risks or discomforts associated with this study, other than the aforementioned inconvenience of time for those selected to be interviewed. Neither you nor the students will be subjected to any sensitive questioning, and there are no costs involved on the students' or your part. Students selected for the interviews may have the advantage of additional one-on-one interaction with a CIS110 instructor. Interviewees will also be excused from one two-hour parameter lab if they choose.

Safeguards will be used to assure your and your students' anonymity and the confidentiality of the data collected. No names will be associated with the recordings, transcriptions, or other data. All data will be kept in a locked office in the Science Building. Statistical results will be reported in the aggregate. If any individual quotes or descriptions are published, they will be disclosed in such a manner as to ensure anonymity and confidentiality, using either general descriptions (e.g., One female student said, "....) or pseudonyms. Results of this study will be made available to the University of Wisconsin-Stevens Point and other associations/publications interested in computer programming instruction.

Your participation and your students' participation in the study are completely voluntary. A decision not to participate will involve no penalty or prejudice on the part of the University of Wisconsin-Stevens Point. Students who choose are free to discontinue participation without penalty or prejudice.

Once the study is completed, I will be glad to share the results with you. In the meantime, if you have any questions please contact:

Ms. Sandra Madison, Assistant Professor of Computing
Department of Mathematics and Computing
University of Wisconsin-Stevens Point
(715) 346-4612
email: smadison@uwspmail.uwsp.edu

If you have any complaints about your treatment as a participant in this study, please call or write either:

Dr. Berri Forman, Consultant	Dr. La Rene Tufts
Institutional Review Board for the Protection of Human Subjects	Institutional Review Board for the Protection of Human Subjects
Environmental Health and Safety	University of Wisconsin-Stevens Point
University of Wisconsin-Milwaukee	Stevens Point, WI 54481
P.O. Box 413	(715) 346-4511
Milwaukee, Wisconsin 53201	
(414) 229-6016	

Although Dr. Forman or Dr. Tufts will ask your name, all complaints will be kept in confidence.

Upon consent, please complete the following:

I understand the components of this study, as outlined in this letter, and agree to participate. I understand that participation in this study is strictly voluntary.

________________________________ Name

__________________ Date

This research has been approved by the University of Wisconsin-Stevens Point and the University of Wisconsin-Milwaukee Institutional Review Boards for the Protection of Human Subjects for a one-year period.

Appendix B

Participant Informed Consent

As an assistant professor in the Department of Mathematics and Computing at the University of Wisconsin-Stevens Point. I am conducting a study of Pascal students' understanding of the construct of parameter passing, one of the pivotal concepts taught in CIS110. I would appreciate your participation in the study, as it may inform our efforts to improve the way we teach the construct.

Upon your consent, you will be given an Inventory of Piaget's Developmental Task (IPDT) and a background survey to complete. They take about 30 minutes total to complete and will be administered during the next class.

In addition, demographic and background information will be collected from the background survey. I will use the information to investigate possible relationships between these variables and your understanding of the construct of parameter passing.

I will observe selected sessions of the class during the semester and may videotape some activities.

Some of you will be asked to participate in a series of three interviews as you study the construct of parameter passing. The interview questions will focus on your computing and programming background and your understanding of the construct of parameter passing. The interviews will be audio tape recorded and transcribed. It is estimated that the interviews will last from 20-30 minutes. The interviews will be conducted outside of class, but every effort will be made to schedule them at your convenience. Interview candidates will be chosen to provide a broad range of prior computing experience and backgrounds.

There are no foreseeable risks or discomforts associated with this study, other than the aforementioned inconvenience of time for those selected to be interviewed. You will not be subjected to any sensitive questioning, and there are no costs involved on your part. Students selected for the interviews may have the advantage of additional one-on-one interaction with a CIS110 instructor. Interviewees will also be excused from one two-hour parameter lab if they choose.

Safeguards will be used to assure your anonymity and the confidentiality of the data collected. No names will be associated with the recordings, transcriptions, or other data. All data will be kept in a locked office in the Science Building. Statistical results will be reported in the aggregate. If any individual quotes or descriptions are published, they will be disclosed in such a manner as to ensure anonymity and confidentiality, using either general descriptions (e.g., One female student said, "....) or pseudonyms. Results of this study will be made available to the University of Wisconsin-Stevens Point and other associations/publications interested in computer programming instruction.

Participation in the study is completely voluntary. A decision not to participate will involve no penalty or prejudice on the part of the University of Wisconsin-Stevens Point. If you choose to participate, you are free to discontinue participation without penalty or prejudice.

Once the study is completed, I will be glad to share the results with you. In the meantime, if you have any questions please contact:

Ms. Sandra Madison, Assistant Professor of Computing
Department of Mathematics and Computing
University of Wisconsin-Stevens Point
(715) 346-4612
email: smadison@uwspmail.uwsp.edu

If you have any complaints about your treatment as a participant in this study, please call or write either:

Dr. Berri Forman, Consultant	Dr. La Rene Tufts
Institutional Review Board for the Protection of Human Subjects	Institutional Review Board for the Protection of Human Subjects
Environmental Health and Safety	University of Wisconsin-Stevens Point
University of Wisconsin-Milwaukee	Stevens Point, WI 54481
P.O. Box 413	(715) 346-4511
Milwaukee, Wisconsin 53201	
(414) 229-6016	

Although Dr. Forman or Dr. Tufts will ask your name, all complaints will be kept in confidence.

Upon consent, please complete the following:

I understand the components of this study, as outlined in this letter, and agree to participate. I understand that participation in this study is strictly voluntary.

Name ______________________ Date ______________

This research has been approved by the University of Wisconsin-Stevens Point and the University of Wisconsin-Milwaukee Institutional Review Boards for the Protection of Human Subjects for a one-year period.

Appendix C

Background Survey

1. What is your gender?	a. male b. female
2. Which category best describes your age?	a. under 21 b. 21-25 c. 26-30 d. 31-40 e. over 40
3. Which phrase best describes the extent of your computer experience before taking this course?	a. never used a computer b. used a computer a few times c. used a computer occasionally d. used a computer extensively
4. Which phrase best describes the nature of your computer experience before taking this course?	a. none b. mostly game playing c. mostly computer aided instruction in other classes (e.g. tutorial, drill) d. mostly word processing, spreadsheets, or other application programs e. mostly programming
5. Which phrase best describes your use of the campus computing facilities, excluding in-class laboratory activities?	a. do not use b. use occasionally c. use regularly d. use extensively e. use exclusively
6. Excluding the student computing labs, how much access do you have to a personal computer?	a. none b. limited c. regular d. extensive e. unlimited
7. If you use the student computing labs, rate the labs in terms of access.	a. almost always available when I need them. b. usually available without waiting c. frequently wait for a work station d. almost always have to wait for a work station e. times when lab facilities are available are not convenient

8. Did you complete a computer awareness or applications course in high school?	a. yes b. no c. uncertain
9. Did you complete a computer awareness or applications class (e.g., CIS101, CIS102 [not programming]) here or at another post-secondary institution?	a. yes b. no c. uncertain
10.Have you completed a one-semester programming course prior to this semester?	a. yes b. no c. uncertain
11.If you completed a one-semester programming course, was the primary language Pascal?	a. yes b. no c. uncertain
12.How much have you used a computer in an employment situation?	a. not at all b. rarely c. occasionally d. regularly e. extensively
13.In which subject do you intend to major?	a. Computer Information Systems b. Business c. Mathematics d. Undecided e. Other
14.In which subject do you intend to minor?	a. Computer Information Systems b. Business c. Mathematics d. Undecided e. Other
15.Which phrase best describes your reason for taking CIS110?	a. required b. recommended by advisor or a friend c. general interest in computing d. undecided about my future and exploring different areas e. other reasons (e.g., scheduling, reputation as an easy "A", etc.)
16.Do you plan to take CIS 111?	a. yes b. no c. undecided

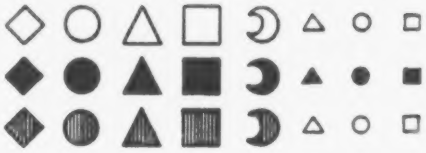
17. Which phrase best describes your career experience?	a. training for my first career b. training for a career change c. considering or training for advancement in my present career
18. Which phrase best describes your prediction about your performance in CIS 110?	a. confident that I will be successful b. hope that I will be successful but have some concerns c. do not expect to be very successfu!, but expect to pass d. concerned that I will not pass the course e. cannot predict
19. Which phrase best reflects your feelings about mathematics?	a. I really enjoy it and do well. b. I do not really enjoy it but I do well. c. I enjoy it but do not do all that well. d. I do not enjoy it and I do not do well. e. Mathematics evokes no particular reaction for me, one way or another.
20. Rate the job potential as you see it for people entering the computing field compared to the general employment market.	a. much better opportunities b. somewhat better opportunities c. no difference d. somewhat poorer opportunities e. much poorer opportunities
21. Rate the job potential as you see it for yourself in the computing field.	a. good potential for me b. some potential for me c. little or no potential although I have some interest in the computing area d. no potential as my career goals are in another area e. undecided

Appendix D

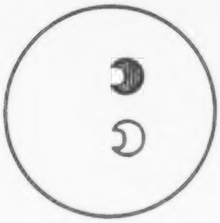
Excerpt from Inventory of Piaget's Developmental Tasks

example

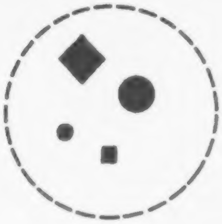
We have many different blocks like these.



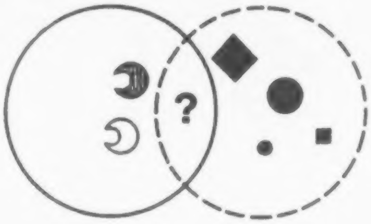
These blocks go in this circle.



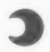
These blocks go in this circle.



We put the circles together like this.



Which block belongs in the middle?



[✓ A]

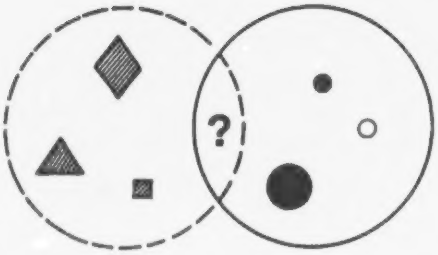
B

C

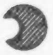
D

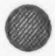
53

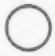
Here are two circles.



Which block belongs in the middle?


A

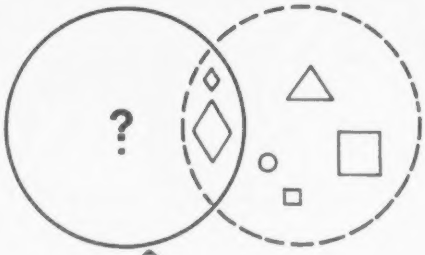

B


C

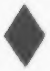
D

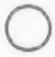
54

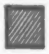
Here are two circles.



Which block belongs in this circle?


A

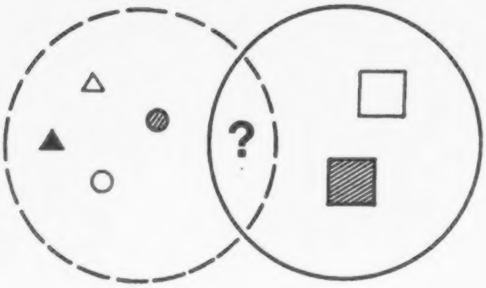

B


C

D

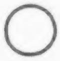
55

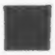
Here are two circles.



Which blocks belong in the middle?

A

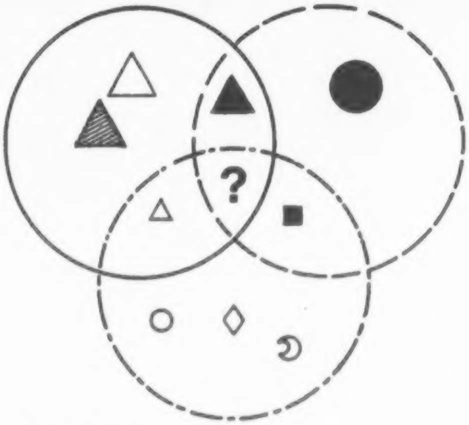

B


C

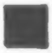
D

56

Here are three circles.



Which block belongs in the middle?


A

B

C

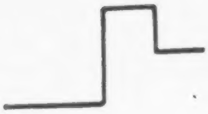
D

59

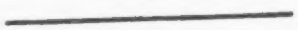
Here are 4 wires.

Which is the shortest?

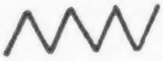
A



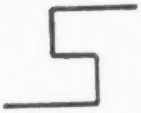
B



C

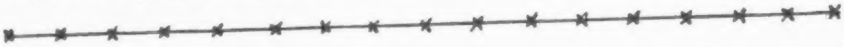


D

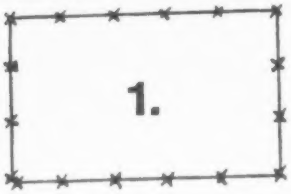


60

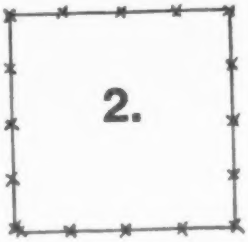
Here is a long piece of barbed wire.



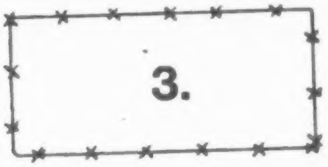
We bend it to make figure 1.



Then bend the same wire to make figure 2.



Then bend the same wire to make figure 3.



Which figure is the largest or are they all the same?

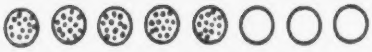
No. 1 largest
A

No. 2 largest
B

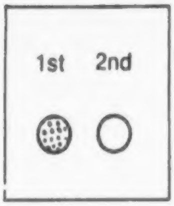
No. 3 largest
C

all the same
D

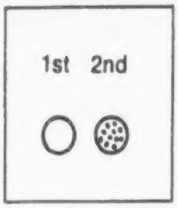
70

We put these balls into a box. 

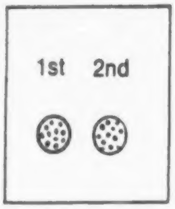
Which balls do you guess you will take out 1st and 2nd ?



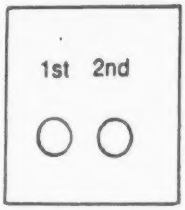
A



B

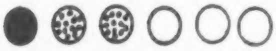


C

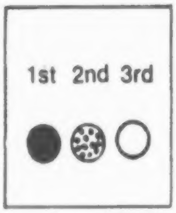


D

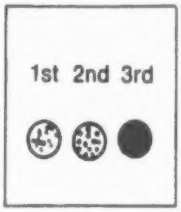
71

We put these balls into a box. 

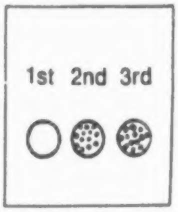
Which balls do you guess you will take out 1st, 2nd, and 3rd?



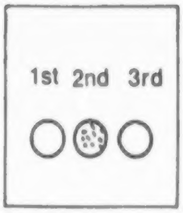
A



B



C



D

72

We put these balls into a box. ● ● ● ◍ ○ ○

Which balls do you guess you will take out 1st 2nd and 3rd?

1st	2nd	3rd
●	○	●

A

1st	2nd	3rd
●	◍	○

B

1st	2nd	3rd
●	○	○

C

1st	2nd	3rd
●	○	◍

D

Appendix E

Laboratories 7 and 9
CIS 110
Laboratory 7

Objectives: To develop skill passing simple parameters
To write an original program that uses a procedure and passes parameters.

Download LAB7A.PAS and LAB7B.PAS from the CIS 110 directory using Courseware Download. You may work with a partner for this part of the lab, but you are not required to do so. Turn the remainder of the program in today if you finish, otherwise next class.

2. Complete the programs following the instructions in the comments included in the program. Test the programs to ensure they work correctly. Turn the program listings and printscreens which demonstrate you tested the programs.

3. Write a program TestFlipper that will call a procedure Flipper. Flipper will exchange the values of the mainline's two variables which will be obtained interactively. The main will display the contents of its variables before the procedure call and after the procedure call. Be sure the WriteLns are explicit enough and complete enough to demonstrate that the parameters were passed correctly (e.g., *WriteLn (First: 6, Second: 6);* would be insufficient.). Turn the program listings and printscreens which demonstrate you tested the programs.

Turn in: Grading Sheet, name side up
Program Listings
PrintScreens of executions
stapled in order.

Grading Sheet
LAB 7

Name(s) ______________________ Section ___________ Score________

1. LAB7A.PAS		
Correct Formal Parameters	5 points	__________
Correct Actual Parameters	5 points	__________
Proper use of value and variable parameters	5 points	__________
2. LAB7B.PAS		
Correct Formal Parameters	5 points	__________
Correct Actual Parameters	5 points	__________
Proper use of value and variable parameters	5 points	__________
Correct Calculations in PROCEDURE	5 points	__________
3. Flipper		
Appropriate Parameter Passing	5 points	__________
Correct results from the procedure	5 points	__________
Driver program tests parameter correctness	5 points	__________
Total 50 points		________________

```
PROGRAM Lab7a;
USES
   Crt, Printer;

{**********************************************
 *  The following procedure returns the       *
 *  average of three values passed to it and  *
 *  informs the calling module of the result  *
 *                                            *
 *  Complete the code by providing both the   *
 *  FORMAL and ACTUAL PARAMETER LISTS         *
 *  You may NOT change any other aspect of    *
 *  the code in either the PROCEDURE or the   *
 *  Main module                               *
 **********************************************}

PROCEDURE AverageThree(                    );
BEGIN
   Num1 := Num1 + Num2;
   Num1 := Num1 + Num3;
   Mean := Num1 / 3;
END;                    {END PROCEDURE AverageThree}

VAR
  First,
  Second,
  Third   :  Integer;
  Average :  Real;

BEGIN                             {BEGIN Main}
  Write(‘Enter an integer variable: ’);
  ReadLn(First);
  Write(‘Enter another integer variable: ’);
  ReadLn(Second);
  Write(‘Enter a third integer variable: ’);
  ReadLn(Third);
  WriteLn(‘Before PROCEDURE call the variables are: ’,First:6,
                 Second:6, ’ and ’, Third:6);

   AverageThree (               );

  WriteLn(‘After PROCEDURE call the variables are: ’,First:6, Second:6,
           ’ and ’, Third:6);
   WriteLn(‘Their average is: ’, Average:8:2);
END.
```

```
PROGRAM Lab7B;
USES
   Crt, Printer;

{**********************************************
 *  The following procedure returns a user    *
 *  entered percent as a decimal value        *
 *                                            *
 *  Complete the code by providing both the   *
 *  FORMAL parameter list and any necessary   *
 *  local variables                           *
 *  You may NOT change any other aspect of    *
 *  the code in the PROCEDURE                 *
 **********************************************}
PROCEDURE GetPercent(              );

BEGIN
   Write(‘Please enter the percent as a whole number: ’);
   ReadLn(Whole);
   Percent := Whole / 100;
END;                    {END PROCEDURE GetPercent}


{**********************************************
 *  For PROCEDURE ProvideResults, complete    *
 *  parameter list as needed.  Account for    *
 *  local variables and do any necessary      *
 *  calculations to produce reasonable results*
 *  DO NOT change any WriteLn statement--you  *
 *  may add statements before or after any    *
 *  existing WriteLn                          *
 **********************************************}
PROCEDURE ProvideResults(                  );

BEGIN
   WriteLn(‘Salary before Taxes: ’, Before:8);
   WriteLn(‘Federal tax rate of ’, FedRate:4:2, ‘ = $’, FedBucks:8:2);
   WriteLn(‘State tax rate of ’, StateRate:4:2, ‘ = $’,
StateBucks:8:2);
   WriteLn(‘City tax rate of ’, CityRate:4:2, ‘ = $’, CityBucks:8:2);
   WriteLn(‘Leaving a take home pay of: $’, NotEnough:8:2);

END;                    {END PROCEDURE ProvideResults}
```

```
{**********************************************
 *  In the Main, provide the appropriate      *
 *  parameters for the PROCEDURE calls.       *
 *  You may also need to provide some         *
 *  additional input code                     *
 *  Change nothing else in the Main           *
 **********************************************}
VAR
  FedTax,
  StateTax,
  CityTax   :  Real;
  GrossWage : Integer;

BEGIN                          {BEGIN Main}
   ClrScr;

   WriteLn(‘Enter the % of Wage paid as Federal Tax’);
   GetPercent(             );

   WriteLn(‘Enter the % of Wage paid as StateTax’);
   GetPercent(        );

   WriteLn(‘Enter the % of Wage paid as City Tax’);
   GetPercent(             );

   Write(‘Enter Salary before taxes: $’);
   ReadLn(GrossWage);

   ProvideResults(              );

END.
```

Appendix E continued

CIS 110
Laboratory 9Instructions
November 18/21

Name ______________________ Section ___________ Score________

1. Download LAB9A.PAS and LAB9B.PAS from the CIS 110 directory using Courseware Download.

2. Complete LAB9A.PAS and test the program to ensure it does what it advertises. Turn the program listing and printscreens which demonstrate you tested all paths of the program. **You must not alter the existing code.**

3. Complete LAB9B.PAS and test the program to ensure it does what it advertises. Turn the program listing and printscreens which demonstrate you tested all paths of the program. **You must not alter the existing code.**

Turn in a disk with your hard copy material. (In a correctly assembled folder)

		LAB9a	Lab9b
Uses Boolean operator correctly	3	____________	____________
Uses Procedures correctly	12	____________	____________
Produces correct output	10	____________	____________
Tests program thoroughly	5	____________	____________

```
{**********************************************************************
*             Author:                                                 *
*             Course: CIS 110                                         *
*             Section:                                                *
*             Date Due:                                               *
*             Instructor:                                             *
*---------------------------------------------------------------------*
* This program reverses a number and displays the reversed number     *
*Ex:  354 --> 453.(1000 -->1 is acceptable-need not print leading 0's)*
***********************************************************************

PROGRAM ReverseNumber;
USES
  Crt;
{**********************************************************************
*             PROCEDURE : GetInteger                                  *
*             Purpose   : Get an integer number to reverse            *
*                         0 < Number <= 9999                          *
*                                                                     *
***********************************************************************
*}
PROCEDURE GetInteger

BEGIN
   Write ('Enter a positive integer with value less than 10,000 ==> ');
   ReadLn (Number);
        {keep entering number until it is greater than 0 and <= 9999}

    BEGIN
      Writeln ('Number must be positive and less than 10,000');
      ReadLn (Number);
    END;
  {endwhile}
   Writeln;
END;   {procedure GetInteger}
```

```
{**********************************************************************
*              PROCEDURE : TurnAround                                  *
*              Purpose   : Reverses the program's variables            *
*                                                                      *
**********************************************************************}
PROCEDURE TurnAround(     Forward: Integer;
                      VAR Backward: Integer);

BEGIN
   Backward := 0;          {Note (Backward*10) causes overflow when }
   WHILE Forward > 0 DO    {Number is 5 digits.                     }
    BEGIN
      Backward := (Backward*10) + (Forward MOD 10);
      Forward := Forward DIV 10
     END;
   {endwhile}
END;                    {procedure Turnaround}

{**********************************************************************
*              PROCEDURE : PrintReverse                                *
*              Purpose   : Prints the number and its reverse           *
*                                                                      *
**********************************************************************}

PROCEDURE PrintReverse (Number,
                        Reverse: Integer);

BEGIN
   Writeln ('Original number =====> ', Number:5);
   Writeln ('Its reverse =========> ', Reverse:5);
END;     {procedure PrintReverse}

{------------------------------Main-------------------------------}
VAR
   Number,
   Reverse: Integer;
BEGIN
  {Procedure call to get a valid integer}
  {Reverse the number}
  {Print the number and its reverses}
END.
```

```
{*******************************************************************
*             Author:                                              *
*             Course: CIS 110
*             Section:
*             Date Due:
*             Instructor:
-------------------------------------------------------------------
*This program counts the number of even and odd numbers and reports
* the count and percentage of each.                                 *
*******************************************************************}
PROGRAM CountEvensAndOdds;
USES
  Crt, Printer;

CONST
  Sentinel = 0;

TYPE
  String4 = STRING [4];

{*******************************************************************
*             PROCEDURE : InitializeCounters                       *
*             Purpose   : Initialize the program's variables       *
*                                                                  *
*******************************************************************}
PROCEDURE InitializeCounters (
BEGIN
  {code here to initialize the NECESSARY Counters}
END;

{******************************************************************
*             PROCEDURE : CalculatePercentage                     *
*             Purpose   : Calculate percentages                   *
*                                                                 *
******************************************************************}
PROCEDURE CalculatePercentage (
BEGIN
  Percentage := Portion / TotalCount
END; {CalculatePercentage}
```

```
{******************************************************************
*             PROCEDURE : DisplayPercentage                      *
*             Purpose   : Display the results of the calculations *
*                                                                *
******************************************************************}
PROCEDURE DisplayPercentage (Message:String4;
                             TotalCount,
                             PartialCount: Integer;
                             Percent: Real;
BEGIN
  WriteLn ('Of the ', TotalCount, ' numbers entered, ', PartialCount,
             ' or ', Percent * 100:3:0, '% were ', Message);

END; {DisplayPercentage}

{******************************************************************
*             PROCEDURE :                                         *
*             Purpose   :to input valid numbers. Reject negative and
*                         numbers over 1000
*******************************************************************
PROCEDURE GetInput (
BEGIN
  Write ('Please enter a whole number. Zero to quit ==> ');
  ReadLn (WholeNumber);
  {add validation code here}
END;  {GetInput}

{******************************************************************
*             PROCEDURE :WriteUserInstructions
*             Purpose   :Tell the user about the program
*
*******************************************************************
PROCEDURE WriteUserInstructions;
BEGIN
  ClrScr;
  Writeln ('This program counts the number of odd and');
  Writeln ('even numbers and calculates the percentage of the');
  WriteLn ('total of each category. Output remains on the screen.');
  Writeln;
END; {WriteUserInstructions}

VAR
  Number,
  OddCount,
  EvenCount,
  Count: integer;
  OddPercent,
  EvenPercent: Real;
```

```
BEGIN
   {Display some user instructions here.}
   {Initialize the variables that need to be initialized here}
   {input a valid number}
  WHILE Number <> Sentinel DO
    BEGIN
     {accumulate the appropriate counters}
     {input another valid number}
    END;
  {endwhile}
  IF Count <> 0 THEN
    BEGIN
      {Calculate the percentage of even numbers}
      {Calculate the percentage of odd numbers}
      {Display the count and percentage of even numbers--call follows}
      DisplayPercentage ('even', Count, EvenCount, EvenPercent);
      {Display the count and percentage of odd numbers}
    END {THEN}
  ELSE
     WriteLn ('No numbers entered');
  {endif}
  WriteLn ('Thank you for using my number program!')
                                  END.
```

Appendix F

Banner and Revised Banner Assignments

The Banner

Objectives:
1. Using programmer-written procedures without parameters.
2. Preparing a structure (hierarchy) chart.
3. Producing aesthetically pleasing output.

Tasks:
Prepare a hierarchy (structure) chart showing the relationship of the various program modules.

Write a properly documented program (using the programming style introduced in class and in the coding standards) to produce a one-word banner. You may choose any five-letter word (minimum) you choose for your banner with one minor consideration--at least three of the letters must be the same. Examples are 'MOMMY,' 'rarer,' 'pepper,' 'sassy,' 'poppy,' or 'MISSISSIPPI.'

Using any keyboard characters you choose (remember that one of the objectives is to produce aesthetically pleasing output), create a banner in this general form.

```
*         *
**       **
* *     * *
*  *   *  *
*   * *   *
*    *    *
*         *
*         *

***********
*         *
*         *
*         *
*         *
*         *
*         *
*********** etc.
```

Write a separate procedure for each block letter. The procedure for "L" might, for instance, be called *PrintL*.

Include some instructions for the user about the purpose of the program. Put these instructions in a programmer-written procedure. *WelcomeUser* or *WriteUserInstructions* or *ExplainProgram* are a few possible procedure names. These instructions always remain on the screen.

Direct the output from the original version of your program to the screen. Include the source code listing of this version in your folder.

Be sure the program includes some comments for the person who reads the Pascal code.

Once you are reasonably satisfied with the appearance of your banner, alter the program to direct the output (banner only) to the printer. Do not direct your output to the printer until you are confident it is working correctly. By doing so, you will save yourself time and effort, conserve paper, and show consideration for your fellow students in the computer lab. Recompile your program and rerun it.

CIS 110 Programming Assignment 6
The Banner

Date Due: _______ Name: ____________________ Section _____ Score ________

Assemble the required materials stapled in order in a properly labeled folder.

1. Grading Sheet
2. Hierarchy/Structure Chart
3. Source code listing with output directed to the screen
4. Source code listing with output directed to the printer
5. Output (banner)

Grading Sheet (this side up)

1. Correct folder, contents, order (required for acceptance)	0 pts.	__________
2. Hierarchy (structure) chart	10 pts.	__________
3. Source code style (follows coding standards and assignment specifications--instructions, comments, etc.)	5 pts.	__________
4. Uses procedures correctly	15 pts.	__________
5. Computer output		
Appropriate user instructions	5 pts.	__________
Produces correct output	10 pts.	__________
Aesthetics (banner appearance)	5 pts.	__________
Total Possible	50 pts.	__________

Appendix F continued

CIS 110 Programming **Bonus Assignment**
The Banner Revised

Date Due: _______ Name: ___________________ Section _____ Score ________

Objectives:

1. Using programmer-written procedures with value parameters.
2. Evaluating whether an existing program is worth modifying.

Task: Modify the program you created for Assignment 6 (or start over if you feel that is easier) to allow the main line (main module, executable portion, statement portion, boss) to determine the character each module will use in forming its block letter and pass that character to the module as a parameter. For the repeated letter in the banner, the mainline module should select a different character each time the letter is displayed/printed. In other words a procedure to print a block letter "L" could produce

```
*              or   #            or   @        depending on the parameter
*                   #                 @        passed.
*                   #                 @
*******             #######           @@@@@@@
```

Submit in the same folder and at the same time as Assignment 6.

Assemble the required materials stapled in order in a properly labeled folder.

1. Grading Sheet
2. Source code listing--either to the screen or printer
3. Output (banner)

Grading Sheet (this side up)

1. Correct folder, contents, order (required for acceptance)	0 pts.	_________
2. Source code style (follows coding standards and assignment specifications--instructions, comments, etc.)	5 pts.	_________
3. Uses value parameters correctly	15 pts.	_________
4. Computer output Appropriate user instructions, correct output, etc.	5 pts.	_________
5. Aesthetics (banner appearance)	5 pts.	_________
Total Possible **Bonus**	30 pts.	_________

Appendix G

Burger Queen Assignment

Date Due:

Objectives: To create a structure diagram, also known as a hierarchy chart, (showing data flow) of an original program.
To produce pseudocode for the main module of a modular program.
To develop an original program that uses procedures with parameters.

Problem: Princess Coal Black (you may have heard of her ancestor Snow White), heir apparent to the renowned Charming family crown, has awarded you the contract to the *Burger Queen* payroll list application program. The system must be ready to produce a payroll list for the work week ending November 8, 199? Princess Co (as she is affectionately known by her contemporaries) 's future depends on your successful implementation of the system.. Hence, as is customary with contracted information systems, you will be penalized for every business day the system is late. Because Princess Co's future depends on the successful implementation of the system, the contract also specifics the penalties to be assessed for failure to accurately follow the specifications. The specifications for the payroll list follow.

Input: Identification number (of the form 9999, which you may assume will be entered correctly), Hours Worked, in half hour increments. (Ex: 35.5 would be 35 and a half hours)
Pay Rate (per hour rate in the form of 99.99).

As the work force in Camelot is somewhat volatile, it is not known in advance how many people will work during the week or indeed if anyone will work. Continue processing until an Identification number of 0 is entered to signal the end of the input.

Processing: Employees are to be paid at a rate of 1.5 times their regular rate for any time over 40 hours. From each employee's gross pay (as calculated above), subtract Cameleot Federal Taxes at a rate of 19% and State taxes at a rate of 4%. Calculate the Gross Pay, Federal Tax, State Tax, and Net Pay (the gross pay less the federal and state taxes) for each *Burger Queen* employee. Because the Charming family fortune is in jeopardy, the monarchs (King and Queen Charming) also need to know the total payroll costs for the *Burger Queen* operation for the week--including the number of employees, total gross pay, total federal tax, total state tax, and total net pay.

Output: An attractively formatted report (King Charming is quite vain about the appearance of the *Burger Queen* reports) that is printed (directly on the printer). The report should effectively advertise *Burger Queen* to any one who reads the report (have a

Burger Queen title), should include appropriate column headings, and should have columns that line up. Report weekly totals in the appropriate columns.

Constraints: King and Queen Charming prefer (for security reasons) that the programming of this project be apportioned among several **(MYTHICAL** because you are going to do this independently**)** programmers. Hence it is to be a well-structured program with each task done by a separate module (procedure). The boss module will do what real world bosses do--give orders and make decisions. The modules also get only the information they need to know to do their jobs.

CIS 110 Assignment 8
Grading Sheet

Name ____________________ Section ______________ Score ______________

No points will be awarded to any program that contains and uses global variables and/or does not use procedures.

Test with:

1. 0 first (should have a report with title, column headings, and 0 totals).

2.

ID	HoursWorked	PayRate
1234	30.0	5.25
5678	56	6.00
9012	39.5	7.50
3456	40	8.00
7890	40.5	9.00
1111	55	10.00

Appendix H

Procedure-related Examination Questions

Examination Date: Date November 11, 19__

1. Which of the following are reasons for using PROCEDURES in Pascal programs?
A. Repeated use of the same task at several points within a program.
B. Need to implement the same task in several programs
C. It simplifies team programming
D. Modularity
E. All of the above are reasons that Pascal employs procedures

2. Which of the following are true of Pascal's passing of parameters (variable) to and from procedures?
A. A value which is input from the keyboard in a procedure is automatically global
B. If a variable modified in a procedure is to have its changed value effective in the calling procedure (an IN-OUT parameter) it should be a value parameter.
C. If a variable modified in a procedure is to have its changed value effective in the calling procedure (an IN-OUT parameter) it should be a variable parameter
D. Pass by value is the same as using global variables
E. None of the above are true

4. In a formal parameter section, variable parameter declarations are immediately preceded by
A. a colon
B. VAR
C. BEGIN
D. a left parenthesis
E. none of the above

6. Values that are ONLY imported (input) into a procedure should be declared parameters.
A. value
B. variable
C. intermediate
D. informal
E. none of the above

For questions 7 - 9, use the following program.
PROGRAM Drill;

```
          PROCEDURE Switch (VAR X, Y :Integer);
          VAR
           Temp:Integer;
          BEGIN
           Temp:=Y;
           Y:= X;
           X:= Temp
          END;          (procedure Switch)
          VAR
           A,
           B,
           C:  Integer;

          BEGIN          {main)
              A:=4;
              B:=6;
              C:=9;
              IF A > B THEN                                   {p2)
                Switch (B, A)
              ELSE
                Switch (A ,B);
              {endif)
              Switch (B, C);
              WriteLn   (A,'      ',B,'      ',C)               {P1}
              END.
```

7. What does line {P1) display?
A. 4 6 9
B. 6 4 9
C. 4 9 6
D. 6 9 4
E. 9 6 4

8. What does line (p 1) display if line (p2) is changed to the following?
IF A < B THEN
A. 4 6 9
B. 6 4 9
C. 4 9 6
D. 6 9 4
E. 9 6 4

9. What does line {P1} display if line {p2} is changed to the following?
IF A < > B THEN

A. 4 6 9
B. 6 4 9
C. 4 9 6
D. 6 9 4
E. 9 6 4

For questions 10- 12, assume the program has returned to its original form. i.e., IF A > B THEN

10. What would line (p1) display if the procedure heading line were the following?
PROCEDURE Switch (X : Integer; VAR Y : Integer);

A. 4 6 9
B. 4 4 4
C. 6 6 6
D. 6 4 4
E. 4 9 9

11. What would line (p1) display if the procedure heading line were the following?
PROCEDURE Switch (VAR X : Integer; Y : Integer);

A. 4 6 9
B. 4 4 4
C. 6 6 6
D. 6 4 4
E. 6 9 9

12. What would line (P1) display if the procedure heading line were the following?
PROCEDURE Switch (X: Integer; Y: Integer);

A. 4 6 9
B. 4 4 4
C. 6 6 6
D. 6 4 4
E. 4 9 4

Question 13-17 refer to the code for PROGRAM Drill. There is no partial credit on the following answers. Each answer must be completely correct to receive any credit.

13. Identify all the formal parameters.

14. Identify all the actual parameters.

15. Identify all the value parameters in the formal parameter list.

16. Identify all the variable parameters in the formal parameter list.

17. Identify any local variables in PROCEDURE Switch

18. Values that are exported (output) from a procedure should be declared as _______ parameters.
A. value
B. variable
C. constant
D. string
E. none of the above

19. Consider the following procedure heading line:
PROCEDURE TestI (A, B: Integer);

Which of the following statements are true?
A. A and B are variable parameters
B. A and B are value parameters
C. A is a value parameter and B is a variable parameter
D. A is a variable parameter and B is a value parameter
E. none of the above is true

20. Actual and formal parameters are associated
A. by position from left to right
B. by exact name match
C. by data type--Integer with Integer, Real with Real and so forth
D. by designation--value with value and variable with variable
E. none of the above the number of formal parameters.

21. The number of actual parameters must be
A. greater than
B. less than
C. equal to
D. less than or equal to
E. It doesn't matter; the compiler uses what it needs.

22. If a procedure changes the content of a value parameter
A. the corresponding actual parameter is changed
B. the corresponding actual parameter is not changed
C. an error will result since a procedure cannot change the value of a value parameter
D. only actual parameters specified as a variable will be changed
E. none of the above

23. A Pascal programmer indicates that a variable is being passed as a variable parameter (i.e., changes made to the variable within a procedure have an effect in the calling module)
A. Preceding the variable name with a VAR in the call to the procedure
B. Preceding the variable name with a VAR in the parameter list of the procedure heading line
C. Preceding the variable name with a VAR in both the call and the procedure heading line
D. Doing nothing. No indication is necessary because all parameter passing in Pascal is done as variable parameter passing unless indicated otherwise.

For Questions 29 - 31, suppose that the MAIN program contains these declarations:

```
VAR
      I, J, K, L, M, N : Integer;
     A, B, C, D:  Real;
```

For each of the following, you are given a procedure header and an invocation of that procedure. Indicate with by circling V those that are Valid and by circling I those that are Invalid. Invalid simply means the compiler would indicate a syntax error had occurred.

I V 29. Header: PROCEDURE P1 (A, B, C : Integer);
Called with: P1 (J, K);

I V 30. Header: PROCEDURE P2 (VAR A, B, C : Integer);
Called with: P2 (A, B, C);

I V 31. Header: PROCEDURE P3 (J, K, L: Real);
Called with: P3 (A, B, C);

42. Which of the following words MUST appear in the Pascal code of any Pascal program using programmer written procedures?

A. VAR
B. PROCEDURE
C. VALUE
D. CALL
E. FUNCTION

Appendix I

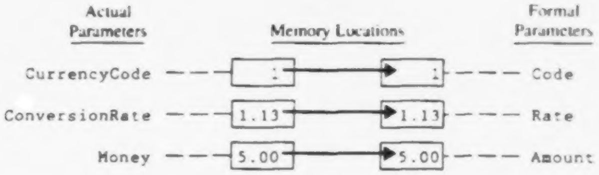
Excerpt from Textbook Coverage of Parameters
(Leestma & Nyhoff, 1993)

Value (In) Parameters

Before a procedure like **Convert** is referenced, its formal parameters are undefined. At the time of reference, memory locations are associated with its value parameters, and the values of the corresponding actual parameters are copied into these locations. Thus, for the procedure **Convert,** the formal parameters **Code, Rate,** and **Amount** are undefined until the procedure reference statement

Convert (CurrencyCode, ConversionRate, Money)

is encountered. At this time, the values of the actual parameters **CurrencyCode, ConversionRate,** and **Money** are copied to **Code, Rate,** and **Amount,** respectively.

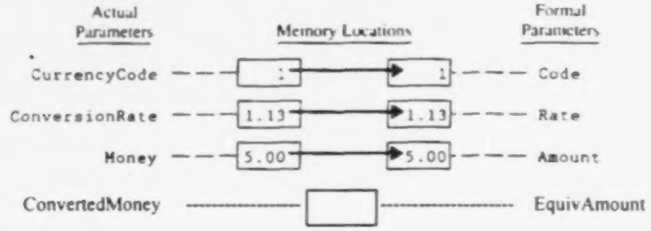


After execution of the procedure, value parameters once again become undefined, so that any values they had during execution of the procedure are lost and are not returned to the main program or subprogram that calls the procedure (p. 189).

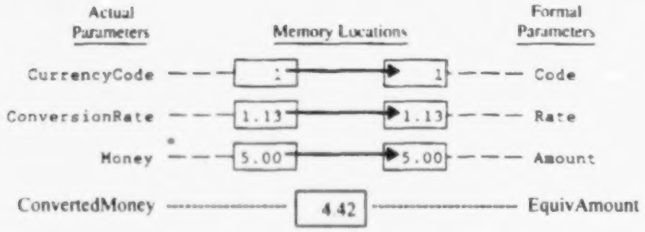
Variable (In-Out) Parameters

As we noted earlier, a procedure must use **variable** or **in-out parameters** to return values to other parts of the program. Thus in the procedure that implements the preceding subalgorithm, the parameter that returns the converted monetary amounts must be a variable parameter (p. 190).

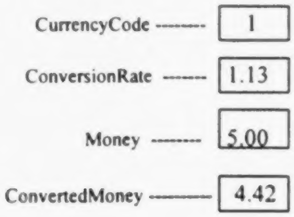
When the procedure reference statement ... is executed, *new memory locations are associated with the formal value parameters* **Code, Rate,** and **Amount** of procedure **Convert2,** and values of **CurrencyCode, ConversionRate,** and **Money** are copied into these locations. No new memory location is obtained for the formal variable parameter **EquivAmount**. Instead, this *variable parameter is associated with the existing memory location of the corresponding actual parameter* **ConvertedMoney**.


Actual Parameters
Memory Locations
Formal Parameters
CurrencyCode
1
1
Code
ConversionRate
1.13
1.13
Rate
Money
5.00
5.00
Amount
ConvertedMoney
EquivAmount

When procedure **Convert2** is executed, a value is calculated for EquivAmount, and because **ConvertedMoney** is associated with the same memory location as **EquivAmount**, this value is also the value of **ConvertedMoney**.


Actual Parameters
Memory Locations
Formal Parameters
CurrencyCode
1
1
Code
ConversionRate
1.13
1.13
Rate
Money
5.00
5.00
Amount
ConvertedMoney
4.42
EquivAmount

Because the memory locations associated with **CurrencyCode**, **ConversionRate**, and **Money** are distinct from those for **Code**, **Rate**, and **Amount**, changes to **Code**, **Rate**, and **Amount** in procedure **Convert2** cannot change the values of **CurrencyCode**, **ConversionRate**, and **Money**. When execution of the procedure is completed, the association of memory locations with **Code**, **Rate**, and **Amount**, and **EquivAmount** is terminated, and these formal parameters become undefined.


CurrencyCode
1
ConversionRate
1.13
Money
5.00
ConvertedMoney
4.42

(p. 194)

Appendix J

Participant Interview One Protocol

Time, date, place

This is Sandra Madison interviewing student ___. Welcome, and thank you for coming. Before we get started, I need to reaffirm that this interview is voluntary and that at any time you may choose not to continue. Do you wish to proceed?

Your instructor will not hear the tape and will not see the transcripts until after the semester ends. There is no way that anything you say will affect your grade. I simply want your honest responses.

Tell me how you came to be enrolled in CIS 110 here at the university.

What experience did you have with computers before taking this class?
(Much, little, games, applications, CAI, etc.?)

What feelings did your prior computer experiences evoke? How did they affect you?
(Intimidating, challenging, fun, easy, engaging, boring, difficult, time-consuming?)

Describe your most memorable computer experience prior to this course.

Tell me about your prior programming experiences.(if any)
What language did you use?

Tell me about your most memorable assignment. Why do you remember this particular assignment?

What about the programming activity did you find most challenging?

How would you define variable? Procedure? Argument? Parameter? Value parameter? Variable parameter?

Thank you for coming. We'll talk again in a few weeks. Next time we'll talk about the Pascal class in which you are currently enrolled.

Appendix K

Participant Interview Two Protocol

Time, date, place

This is Sandra Madison interviewing student ___. Welcome, and thank you for coming. Once again, I need to reaffirm that this interview is voluntary and that at any time you may choose not to continue.

There is no way that anything you say will affect your grade. I simply want your honest responses.

Earlier, we talked about the computing (and programming) experiences prior to this class. By any chance, did the previous interview provoke any computing memories beyond what we have already talked about?

How do you feel about the class so far this semester? Easy, challenging, fun, boring,? Can you explain why?

Describe one experience that stands out in your mind for some reason.

Which assignment was your favorite? Least favorite? Why?

Which activity have you enjoyed the most? The least?

Which concept or topic have you found to be the most challenging so far this semester?

Do you know why it has challenged you the most?

How would you define variable? Procedure? Argument? Parameter? Value parameter? Variable parameter?

Consider the following analogy:

Situation one. There is a class of novice painters, learning to paint. During one of their classes, the instructor brings in a first-class replica of the Mona Lisa for the students to use as a model. The students are instructed to reproduce the painting. The students complete the assignment. They take their copies home, and the instructor returns the model to the supply closet.

Situation Two. The original Mona Lisa painting has deteriorated. It needs some restoration work. The original is delivered to a restoration master who completes the work needed. The restored Mona Lisa is then returned to its owner.

How is this situation analogous to parameter passing in Pascal?

Here is a program, and here is the output it produces. Read it over and decide which of the procedure heading lines would produce the output shown. Then tell my why you think it is so. (Give participant a paper copy of Task 1 and a pencil.)

Thanks for coming. We'll talk some more soon.

Appendix L

Task 1

```
PROGRAM ProcedureExample;

PROCEDURE Swap (     N1: Integer;             {1}
                 VAR N2: Integer);

PROCEDURE  Swap (    N1,                      {2}
                     N2 : Integer);

PROCEDURE Swap (VAR N1,                       {3}
                     N2 : Integer);

PROCEDURE Swap (VAR N1: Integer;              {4}
                    N2: Integer);
.................................................
VAR
  T: Integer;

BEGIN
  T := N1;
  N1 := N2;
  N2 := T
END;    { procedure swap}
```

The output for this program is: 16   16

Which of the above procedure heading lines produces this output? Explain how you know.

PROCEDURE Swap {__________}

```
{main}
VAR
  I,
  J : Integer;

BEGIN
  I := 16;
  J := 5;
  Swap (I, J);
  WriteLn (I:5, J:5)
END.
```

Appendix M

Participant Interview Two Protocol

Time, date, place

This is Sandra Madison interviewing student ___. Welcome, and thank you for coming. Once again, I need to reaffirm that this interview is voluntary and that at any time you may choose not to continue. There is no way that anything you say will affect your grade. I simply want your honest responses.

Repeat if appropriate.
How would you define variable? Procedure? Argument? Parameter? Value parameter? Variable parameter?

Do you see any connection between variables and parameters?

Today I have some tasks for you to solve, and I want to record your thoughts as you solve them. Think aloud as you are solving the problem. Take your time. There is no hurry.

The first task is to complete the procedure heading lines for this otherwise complete program. Think aloud as you solve the problem. (Give participant a paper copy of Task 2.)

Next, I have loaded a one-module program for you. Your task is to modify the program so that it uses procedures and parameters. Each time you compile a version, I want you to save it under a new name (Version1, Version2, etc.) so we have the intermediate versions. Explain each change you make, and tell me what you are thinking as you make the change. (Give participant a paper copy of the program listing for Task 3, which is also loaded into the Turbo 7.0 Pascal editor on my office computer.)

The last task I have is an analysis exercise. Would you look over this program and tell me what the output will be? Think aloud as you're solving the problem. Probe: Why do you think that is the answer? (Give participant a paper copy of Task 4.)

Thank you for your time and cooperation this semester. It has really helped. Once I've finished collecting data, I'll be happy to talk over any of these tasks with you.

Appendix N

Task 2

Complete the missing procedure heading lines so that this program will compile and do the job implied by its name. Remember that the formula for calculating the volume of a rectangular solid is: volume = length x width x depth. As you complete each procedure heading line, explain as completely as you can why you constructed it as you did.

```
PROGRAM CalculateVolumeOfARectangularSolid;
...............................................
PROCEDURE


BEGIN
  Volume := Len * Wid * Dep
END;
....................................................................
PROCEDURE


BEGIN
  WriteLn (‘The volume of the solid is ‘, Answer : 8 : 2)
END;
....................................................................
PROCEDURE


BEGIN
  Write (‘Please enter a dimension ==> ');
  ReadLn (Dimension)
END;
{main}..................................
VAR
  Volume  : Real;
  Length,
  Width,
  Depth:    Real;

BEGIN
  GetInput (Depth);
  GetInput (Width);
  GetInput (Length);
  CalculateVolume (Length, Width, Depth, Volume);
  DisplayOutput (Volume)
END.
```

Construct a set of reasonable inputs and corresponding output for the program.

Appendix O

Task 3

```
PROGRAM PayrollPreparation;
USES   Crt;

CONST
  InputPrompt2 = ‘Please enter hours worked (1/2 hour increments) => ’;
  InputPrompt3 = ‘Please enter pay rate (2 decimal places) => ’;

{main}
VAR
  PayRate,
  HoursWorked,
  GrossPay    : Real;

BEGIN
  ClrScr;
  WriteLn (‘This program computes the gross pay for one employee’);
  WriteLn (‘Overtime is not taken into consideration’);
  WriteLn;
  Write (InputPrompt2);
  ReadLn (HoursWorked);
  Write (InputPrompt3);
  ReadLn (PayRate);
  GrossPay := HoursWorked * PayRate;
  WriteLn;
  WriteLn (‘Hours Worked  ’: 12, ‘Rate   ’:10, ‘Gross Pay  ’:10);
  WriteLn (HoursWorked:12:2, PayRate:10:2, GrossPay:10:2);
END.
```

Appendix P

Task 4

```
PROGRAM ABC;

PROCEDURE Increment (VAR A,
                         B: Integer);
BEGIN
  A := A + 1;
  B := B + 1
END;                   {procedure Increment}

VAR
  N: Integer;

BEGIN
  N := 3;
  Increment (N, N);
  WriteLn (N)
END.
```

Appendix Q

Instructor Interview One Protocol

Time, date, place

This is Sandra Madison interviewing Dr. Greene. Do you mind if we tape record the conversation?

Tell me how you came to be a computer programming instructor.

You taught in another discipline before you taught computing. Can you tell me what attracted you to this subject?

How did you learn to program?

Tell me about your most memorable programming experience.

If you can recall, as a novice, what programming concept or skill did you find most difficult to learn?

You have taught CIS 110 several times in the last few years. Describe it in your own words.

What are your goals for the class? For the students?

Tell me about the CIS 110 lesson that pleased you most.

What about the lesson pleased you least?

What concept seems to give the novice Pascal programmers the most difficulty?

Describe the way you teach the construct of procedures and parameter passing.

How do your students demonstrate their understanding of the construct of parameter passing?

Thanks for talking with me today. We'll talk once more later in the semester.

BEST COPY AVAILABLE

Appendix R

Instructor Interview Two Protocol

Time; date; place

This is Sandra Madison interviewing Dr. Greene. Do you mind if we tape record the conversation?

You have completed the unit on procedures. I'd like to talk about that.

How do you feel this portion of the course went this semester?

What are your perceptions of your students' understanding of the construct of procedures and parameter passing?

What gave them the most difficulties?

Describe the errors or misconceptions they demonstrated, either in class, on the assignments, or on the tests.

Of the different instructional strategies you employed this semester, which do you think was the most helpful in developing your students understand the constructs of procedures and parameter passing? Can you give me an example of why you think this?

Was there a strategy that flopped? What makes you think so?

Is there anything you would like to add for the record that we haven't touched upon during the three interviews?

Thank you for your help this semester. You know how much it means. Hopefully, we will be able to describe our students' understanding of the construct of parameter passing. Then, perhaps we can identify some instructional strategies that will increase that understanding.

Appendix S

Payroll Program Simulation

Objective: To provide a concrete representation of the copy versus shared memory-location perspective of the parameter mechanism.

Context: PayRoll Program (something familiar and concrete). Omits other concept such as loops and selections and allows student to focus only on the construct in question.

Reading: see Appendix T, for example.

Participants: One volunteer for each worker module plus an employee.

Materials Needed:
Variables: Transparency sheets folded and taped to form envelopes for the variables (storage locations). Main module variable name and data type should be permanently applied to one side of the envelope (to simulate the fixed variable name). One for each of the main module variables, plus any necessary for the copies that are passed to the modules.

Values: Card stock cut to the size of the envelope. They can be slid in and out (to simulate the fact that the values in a variable can be changed). Laminated gold for the main module variables provides opportunity for levity and comments such as the "boss module" sharing its valuable information. Non-laminated green or blue card stock for the copies sent to the procedures as value parameters allows for comments such as "dull green copy of the real thing."

Titles for the Modules: Laminated Card Stock that could be taped to the wall, like the title on the door of an office for the boss module and the worker modules (to simulate the procedure names).

Tools: A marking pen for each module

Chits: One for each main module variable, inscribed with a message approximating "Permission to change variable _________ (the main module variable name)" (to simulate passing a variable parameter).

Scripts: Each module needs a short script so that he or she knows what to do.

For example, when control is transferred to WriteUserInstructions, the module should stand up and announce the purpose of the program.

GetInput should ask the person designated as the employee for his or her name, hours worked, and pay rate.

CalculatePay should perform the calculations.

DisplayOutput should announce (and display) the results of the payroll calculation to the audience.

Procedure: The class decides which modules will be invoked, which variables will be passed as parameters to each module, and whether the parameters will be value or variable.

If it's a variable parameter, the "boss" hands the module the chit that gives permission to the module to change the main module (boss) variable.

If it's value, she makes a dull, green, copy of the gold value, places it in one of the spare envelopes, and hands it to the module. The value parameters are tossed into the "bit bucket" (the waste basket) at the module's conclusion.

I ask for volunteers; if no one volunteers I volunteer someone. That person chooses his or her "job" and volunteers the next person.

Reflection: Possible discussion:
- Similarities and the differences between the simulation and the actual process
- Connection to the Pascal syntax (see Appendix U for an example)
- Reasons for the value-variable choices

Extension: What if . . .? questions, perhaps relating to what would have happened had inappropriate choices been made (especially if the class chose everything correctly during the simulation), possibly redoing parts with incorrect choices, encouraging the modules to "clobber" inappropriately VARed parameters.

Appendix T

Revised Leestma and Nyhoff Excerpt

Excerpts from Chapter 5 in our text: *Turbo Pascal: Programming and Problem* Solving by Leestma and Nyhoff (1993), modified to use the Payroll example used in the simulation.

Simple Procedures

The simplest procedures are those that do not receive any information from other subprograms or from the main program and do not return any information to them .... The example to display instructions in the **PayRollCalculation** program does not require information from other parts of the program (p. 175).

```
PROCEDURE WriteUserInstructions;
BEGIN
  ClrScr;
  Writeln ('This program computes the gross pay for one employee');
  Writeln ('Overtime is not taken into consideration');
  Writeln
END; {procedure WriteUserInstructions}
```

Referring to the formal parameters used by the other procedures:

What is needed instead are special kinds of variables in a procedure to which values can be passed from outside the procedure. In Pascal, these special variables are called **formal parameters** and are declared in the procedure's heading. Parameters that can store values passed to them but that cannot return values are called **value parameters** or **in parameters** and are the kind of parameters we consider first....Parameters that can both receive values and return values are called **variable** or **in-out parameters** and are considered later in this section (p. 183).

Value (In) Parameters

```
PROCEDURE DisplayOutput(EmployeeName: String20;
                        Time: Integer;
                        RateOfPay,
                        Pay: Real);
BEGIN
  Writeln;
  Writeln (' Employee':20, 'Hours': 10, '  Rate':10, 'Gross Pay':15);
  Writeln (EmployeeName:20, Time:10, RateOfPay:10:2, Pay:15:2);
END; {procedure DisplayOutput}
```

Before a procedure like **DisplayOutput** is referenced (called), its formal parameters are undefined. At the time of reference, memory locations are associated with its value parameters, and the values of the corresponding actual parameters are copied into these locations. Thus, for the procedure **DisplayOutput,** the formal parameters **EmployeeName, Time, RateOfPay,** and **Pay** are undefined until the procedure reference statement (call).

DisplayOutput (Name, HoursWorked, PayRate, Salary);

is encountered. At this time, the values of the actual parameters **Name**, **HoursWorked, PayRate,** and **Salary** are copied to **EmployeeName, Time, RateOfPay, and Pay** respectively.

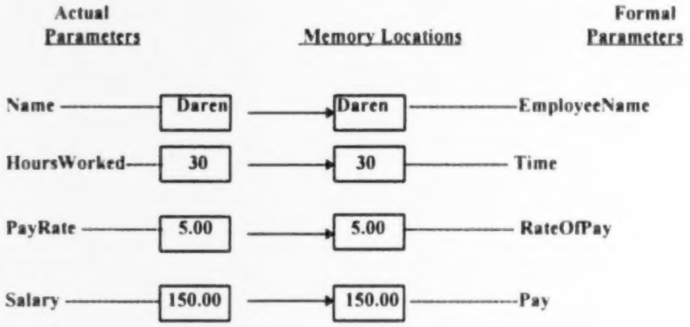


Figure 3. Depiction of parameters during execution of DisplayOutput

After execution of the procedure, value parameters once again become undefined, so that any values they had during execution of the procedure are lost and are not returned to the main program or subprogram that calls the procedure (p. 189).

Variable (In-Out) Parameters

As we noted earlier, a procedure must use **variable** or **in-out parameters** to return values to other parts of the program. Thus in the procedure that implements the preceding subalgorithm, the parameter that returns the converted salary amounts must be a variable parameter. Declarations of formal variable parameters are preceded by the reserved word **VAR** to specify that they are variable rather than value parameters (p. 190).

When a procedure reference statement

CalculateSalary (HoursWorked, PayRate, Salary);

is executed, *new memory locations are associated with the formal value parameters* **TimeWorked** and **RateOfPay** parameters of procedure **CalculateSalary,** and values of **HoursWorked** and **PayRate** are copied into these locations. No new memory location is obtained for the formal variable parameter **GrossPay**. Instead, this *variable parameter is associated with the existing memory location of the corresponding actual parameter* **Salary**.

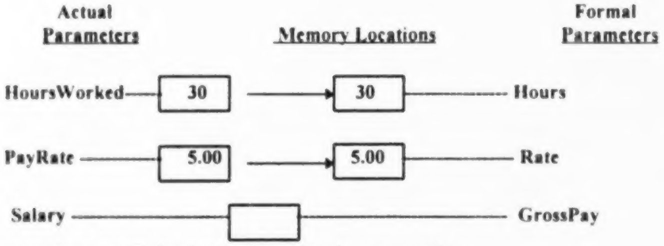

Actual
Parameters
Memory Locations
Formal
Parameters
HoursWorked
30
30
Hours
PayRate
5.00
5.00
Rate
Salary
GrossPay

Figure 4. Depiction of the parameters as CalculateSalary begins execution.

When procedure **CalculateSalary** is executed, a value is calculated for **GrossPay**, and because **Salary** is associated with the same memory location as **GrossPay**, this value is also the value of **Salary**.

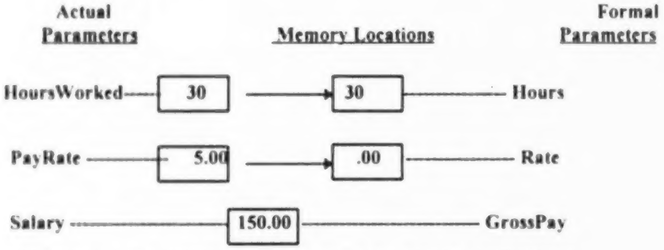

Actual
Parameters
Memory Locations
Formal
Parameters
HoursWorked
30
30
Hours
PayRate
5.00
.00
Rate
Salary
150.00
GrossPay

Figure 5. Depiction of the parameters as the execution of Calculate Salary completes.

Because the memory locations associated with **HoursWorked** and **PayRate** are distinct from those for **Hours** and **Rate**, changes to **Hours** and **Rate** in procedure **CalculateSalary** cannot change the values of **HoursWorked** and **PayRate**. When execution of the procedure is completed, the association of memory locations with **Hours, Rate,** and **GrossPay** is terminated, and these formal parameters become undefined (p. 194).

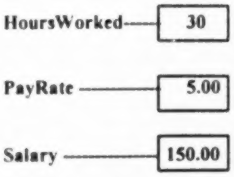

HoursWorked
30
PayRate
5.00
Salary
150.00

Figure 6. Depiction of parameters after execution of CalculateSalary.

Rules for Parameter Association

The following rules summarize the relation between actual and formal parameters:

1. There must be the same number of formal parameters as there are actual parameters.
2. The types of associated formal and actual parameters must agree; however, an actual parameter of **integer** type may be associated with a formal *value* parameter of **real** type, but not with a formal *variable* parameter.
3. An actual parameter associated with a *variable* formal parameter must be a variable; it may not be a constant or an expression (p. 195).

Appendix U

Payroll Program

```
PROGRAM PayrollPreparation;
USES   Crt;

TYPE
  String20 = string[20];

PROCEDURE WriteUserInstructions;
BEGIN
  ClrScr;
  Writeln ('This program computes the gross pay for one employee');
  Writeln ('Overtime is not taken into consideration');
  Writeln
END; {procedure WriteUserInstructions}

PROCEDURE GetInput (VAR WorkerName: String20;
                    VAR TimeWorked: Integer;
                    VAR HourlyRate: Real);
CONST
  InputPrompt1 = 'Please enter the employee''s name ==> ';
  InputPrompt2 = 'Please enter hours worked (1 hour increments) ==> ';
  InputPrompt3 = 'Please enter pay rate (2 decimal places) ==> ';

BEGIN
  Write (InputPrompt1);
  ReadLn (WorkerName);
  Write (InputPrompt2);
  ReadLn (TimeWorked);
  Write (InputPrompt3);
  ReadLn (HourlyRate);
END; {procedure GetInput}

PROCEDURE CalculateSalary (    Hours: Integer;
                               Rate: Real;
                           VAR GrossPay: Real);
BEGIN
  GrossPay := Hours * Rate
END;  {procedure CalculateSalary}
```

```
PROCEDURE  DisplayOutput (EmployeeName: String20;
                          Time: Integer;
                          RateOfPay,
                          Pay: Real);
BEGIN
  Writeln;
  Writeln (' Employee':20, 'Hours': 10, '  Rate':10, 'Gross Pay':15);
  Writeln (EmployeeName:20, Time:10, RateOfPay:10:2, Pay:15:2);
END; {procedure DisplayOutput}

{main}
VAR
  Name : String20;
  PayRate : Real;
  HoursWorked: Integer;
  Salary  : Real;

BEGIN
  WriteUserInstructions;
  GetInput (Name, HoursWorked, PayRate);
  CalculateSalary (HoursWorked, PayRate, Salary);
  DisplayOutput (Name, HoursWorked, HoursWorked, Salary)
END.
```

Appendix V

Task 4 Revised

```
PROGRAM ABC;

PROCEDURE Change (VAR A,
                      B: Integer);
BEGIN
  A := A + 2;
  B := B + 1;
  WriteLn ('A = ', A, '  B = ', B)
END;

VAR  {main}
  N: Integer;
BEGIN
  N := 3;
  Change (N, N);
  WriteLn ('N = ', N)
END.
```

CIS110 Understanding Parameters

Name ____________________________ Section _________

Objectives: To understand how information is communicated among program modules
To be able to explain what causes the difference in the behavior of value and variable parameters.

Individual Activity:
1. Trace through the following program, and predict the output of the WriteLn.
Prediction: _____. Justify your answer.

2. Imagine (or pencil in) the program changed so that both parameters are value parameters. Repeat activity 1. Prediction: ________. Justification:

3. Imagine (or pencil in) the program changed so that A is a value parameter and B is a variable parameter. Repeat activity 1. Prediction: _______. Justification:

Small group activity: Compare your predictions and the reasoning provided for the prediction. Attempt to come to a consensus.

1. Both variable parameters: Prediction: _______. Justification:

2. Both value parameters. Prediction: _______. Justification.

3. A is a value parameter and B is a variable parameter. Prediction: _____. Justification:

Individual Lab Activity:
1. Type in the program (in its original form). Add a ClrScr if you wish.

2. Size the edit window, open a watch window, and move the watch window so you can see both windows completely.

3 Add watches for N, A, and B.

4. Begin tracing at the program's first instruction, using <F7> . Remember that the highlight bar is located at the next instruction to be executed.

Group Lab Activity: (Compare your results and your explanations with those of your neighbors.)

1. (Both VAR parameters) Describe what happened to the "watched" variables as you executed the instruction N := 3.

Explain your findings.

Describe what happened as you executed the instruction A := A + 2.

Explain your findings.

Describe what happened as you executed the instruction B := B + 1.

Explain your findings.

2. Change the program to both A and B value parameters. Repeat the trace. Describe what happened to the "watched" variables as you executed the instruction N := 3.

Explain your findings.

Describe what happened as you executed the instruction A := A + 2.

Explain your findings.

Describe what happened as you executed the instruction B := B + 1.

3. Change the program to make A a value parameter and B a variable parameter. Repeat the trace. Describe what happened to the "watched" variables as you executed the instruction N := 3.

Explain your findings.

Describe what happened as you executed the instruction A := A + 2.

Explain your findings.

Describe what happened as you executed the instruction B := B + 1.

Explain your findings.

Understanding Parameters

Date Due: ____________________ Section __________ Score ______

Name ______________________ Name ______________________

Name ______________________

Write a conjecture about the way variable parameters communicate information among the program modules.

Write a conjecture about the way value parameters communicate information among the program modules.

VITA

Title of Dissertation: A Study of College Students' Construct of Parameter Passing: Implications for Instruction

Full Name: Sandra Kay Madison

Place of Birth: Gary, Indiana

Colleges and Universities:

University of Wisconsin-Milwaukee
1958-1964 Bachelor of Science

University of Wisconsin-Stevens Point
1983-1986 Master of Education-Professional Development

Professional Positions Held: Assistant Professor of Mathematics and Computing

Membership in Honorary Societies:

Sigma Epsilon Sigma	1959
Delta Chi Sigma	1963

Publications: (Proceedings)

The Construct of Parameter Passing: A Study of Novice Programmers. National Education Computing Conference. Baltimore, MD. June 1995.

Creating a More Equitable Computing Climate. American Association of University Women Pre-Conference Symposium. Orlando, FL. June 1995.

Using Student Experience to Guide Curriculum in Introductory Computing National Educational Computing Conference. Orlando, FL. June 1993.

Formal Thinking and Computing Students: Preliminary Results. Midwest Computer Conference. Whitewater, WI. March 1993.

Major Department: Curriculum and Instruction

Minor: Business Administration

Signed Henry S Kepner Jr
Major Professor in Charge of Dissertation